

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



—— 杭建 / 著

# Java工程师修炼之道

为入门者条分缕析地梳理Java技术体系 串联后端关键技能供有经验者实战进阶



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
http://www.phei.com.cn

## 作者简介

---

### 杭 建

重度Java使用者，具有近10年的Java后端开发经验，一直专注于Java EE、系统架构、大数据等后端技术。现任随身云（中华万年历）技术总监，负责公司的技术培训、系统架构、研发管理等工作，带领研发团队完成了大数据平台、推荐系统、广告平台、传媒平台等系统，以及分布式调度、应用性能监测等基础框架的开发，支撑起了中华万年历、微历、牛哞的对话等高达3亿多用户访问量的应用。作者之前曾就职于网易杭州研究院从事基础平台、云计算相关技术的开发工作，参与了易信公众平台、网易云计算动态负载均衡等项目的研发。



# Java工程师修炼之道

杭建 / 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

## 内容简介

Java 开发一直是当前互联网领域最火热的开发技能之一,Java 工程师也一直是需求量非常大的开发职位。本书主要针对一名合格的 Java 工程师的必备技能做了大纲性的总结和阐述。本书内容包括了工程化、常用开发框架、数据存储、数据传输、Java 编程高级知识、性能优化、安全技术等内容,基本涵盖了 Java 工程师需要掌握的绝大部分技能点。

本书可以看作一本 Java 工程师的入职指南,也可以看作一本串联 Java 后端技能点的参考手册。通过精心编排的内容,刚入门的 Java 工程师能够体系化地学习相关开发技能,有经验的 Java 工程师能够查漏补缺,巩固自己的相关开发技能,进一步完善自身的 Java 技术体系。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

## 图书在版编目(CIP)数据

Java 工程师修炼之道 / 杭建著. —北京: 电子工业出版社, 2018.3

ISBN 978-7-121-33501-3

I . ① J… II . ① 杭… III . ① JAVA 语言 - 程序设计 IV . ① TP312.8

中国版本图书馆 CIP 数据核字 (2018) 第 010963 号

策划编辑: 张春雨

责任编辑: 付 睿

印 刷: 三河市双峰印刷装订有限公司

装 订: 三河市双峰印刷装订有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 25.25 字数: 541 千字

版 次: 2018 年 3 月第 1 版

印 次: 2018 年 3 月第 1 次印刷

定 价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式: (010) 51260888-819, [faq@phei.com.cn](mailto:faq@phei.com.cn)。

# 推荐序一

Hey！新来的读者，为了吸引你的注意力我真是煞费苦心，但最终还是没能写出一句特别吸引眼球的话来，毕竟写序的我不是标题党出身。此刻我真的非常能理解你拿到新书之后那渴望知识的心情，所以你恨不得一个字的“序”也不要看到，直接到达“最有价值”的知识点。但作为一名资深转业码农（对！你没看错，是“转业”，不是“专业”）还是想说一句，你先看完序，5分钟后到达知识的战场，会更稳！

相信你已经在看“序”了，那么我们来说点正经事。

你的知识体系的养成有3个关键阶段：看山是山，看山不是山，看山还是山。本书的适用人群是“看山不是山”的那些人，如果你恰好处于这个阶段，恭喜你！书钱没白花。

Java 是一门非常容易入门的语言，初学者经过初期的学习之后基本能掌握 DEMO 级别的编程应用。相信读者你已经度过了这个阶段，但是 Java 庞大的体系可能会把你绕晕，又或者你还没看到 Java 的生态系统有多么复杂。此时，你需要本书。从事程序员这个工作，到比较高阶的时候，其实是不挑语言的，语言只是工具，而你可以在纷繁复杂中游刃有余。但几乎每一位高手都是先深入一个领域，再横向发展的。你可以不用着急后续的横向发展，先坚定自己学习 Java 的信心！因为，从广泛的应用场景、顶级的开源生态、未来可期的薪水和职位来说，Java 都是非常不错的选择。

敲黑板，画重点！下面来解释一下，为什么本书面向的是“看山不是山”的人群。在度过 Java 的入门期之后，会有一个烦恼，那就是面对 Java 这么庞大的体系，我们究竟应该学习什么？选择方向，往往比努力更重要！是使用 J2SE 编写桌面程序？是使用 J2ME 编写嵌入式应用？还是使用 J2EE 编写企业级应用？这些是我们那个泛黄的年代特有的烦恼。而现在的烦恼可能是学 Android？还是学 Java 后端？即便大方向你已经十分坚定，而且选择

了 Java 后端编程，但因为复杂的知识体系和 Google 发布的各种教程文档，眼前看到的已经不再是清晰的山脉，而是一片迷雾。此时，你需要本书，因为它给你指明了努力的方向。

本书的结构、阐述的方式和大部分的“指南”书籍有较大的区别，本书是以笔记和要点的形式进行呈现的，用现在的话说就是捞干货。本书涵盖的知识，是以现代工程实践中的实际案例出发来组织的，所以知识点范围非常广泛，每一个点都对最关键的“Best Practice”简明扼要地进行了说明。你在阅读本书的时候需要一些相关经验，不然无法跟上作者的节奏，建议在有一定的知识准备后再阅读本书，这样你会受益匪浅。从另外一个角度看，在你有了一定的基础积累之后，本书可以帮助你全面地了解一个现代化的最先进的工程实践是怎样的。本书讲述了目前行业中最常用的，经过了实践的工程方案，这将是快速进阶的最佳指引。

——孙建，随身云（中华万年历）联合创始人 & CEO

## 推荐序二

2013 年，我和本书作者的接触是从基于网易的一个大型互联网应用合作开始的，我见证了从第一行代码到整个系统服务于亿级用户的过程，并且相信这种经历对开发者来说是一笔巨大的财富，其中大量的开发和实战经验都会在本书中得到充分的体现，相信读者能从书中直接领略到丰富的实战知识。在与本书作者的合作过程中，其对 Java 技术的热爱与追求，对问题刨根问底，直到理解透彻、灵活应用，这些都令我印象深刻。这些年，我与本书作者一直保持沟通交流、相互学习，他将近十年的实战经验沉淀于本书以实现对外端技术的探索、布道，非常值得开发者与近高窗卧听秋。

后端技术涉及内容非常广泛，Java 语言也是互联网开发行业使用的主流语言，相信后续也将继续流行很长一段时间，而本书作者也一直从事 Java 后端开发工作。在本书中作者比较系统地从总体上描述了后端技术相关的理论知识，包括基础设施、网关服务及框架选型等基本原则，然后以实际经验进行示例说明，接着详细梳理了 Java 的后端技术，相信读者读完本书后会更全面地理解后端技术。互联网的业务建设需要不同角色的开发者共同协作完成，因此，系统工程化是开发者首先要共同遵守的规范或约定，包括代码规范、版本管理和代码质量检查等。

开发框架的选型进一步地为工程化提供了基础，也能加速推进互联网开发，尽管是否重复造轮子是一个恒久的话题，但是没有永远的银弹，只要在合适的时间里根据团队的能力选择合适的技术框架就好。一般来讲，目前常用的框架包括基本的依赖注入、AOP、事务管理、连接池管理、数据操作、日志服务等，在众多的框架中，本书作者选用目前在 Java 领域使用最广泛的 Spring 做深入的分析，详细地说明各组件的基础知识、基本原理和实际使用案例，最难得的是把较多开发者遇到的坑都用真实的示例进行了说明，可以帮助开发者快速地跳过这些伤心地带，同时也把最佳实践画龙点睛地带给开发者。



数据存储无疑是所有系统应用中非常重要的一环,应用的场景用例也和数据库的选型有着极其重要的关系,开发者选择关系型数据库还是非关系型数据库是需要根据软件成本与人力成本来进行权衡的,比如选择 MySQL、Oracle 等开源或商业的数据库。本书重点从数据库的基础知识、索引和表优化等方面以详尽的示例为更好地选择数据库的存储类型提供了更多的知识。

早期的关系型数据库一般能满足数据达到一定规模的企业的需求,而在互联网业务领域,特别是移动互联网领域内的元数据或者日志数据等,达到亿数量级别是很常见的,这时通常使用非关系型数据库,在非关系型数据库里使用非常多的有 MongoDB、HBase 等分布式数据库系统。作者在自身的企业开发实践中,得到了大量的使用经验和最佳实践。为了加速后端应用,缓存热数据是加速业务、提高业务性能、提升用户体验的重要手段,通过使用本地缓存、远程缓存进行数据加速、数据预热或提高数据的命中率,是开发者在应用开发的过程中常会遇到的场景。

“路漫漫其修远兮,吾将上下而求索”,后端技术每年都在不断发展,所用技术也有变化,近些年 Java 语言的发展速度不那么快了,但是总体是在不断前进发展的,本书作者带领的团队一直深耕此领域并希望通过本书为技术开发人员带来更多帮助。

——尧飘海,网易云基础服务(蜂巢)首席架构师



# 前言

目前互联网行业如火如荼，进入这个行业的技术人员也越来越多。对于研发来说，从工程角度其主要分为前端工程师、客户端工程师（又分为 iOS 和 Android 工程师）、后端工程师、算法工程师等职位。本书所说的 Java 工程师指的是以 Java 作为主要开发语言的后端工程师。

笔者从 2008 年还未毕业时做一些小的项目至今，做后端开发已经有差不多 10 年时间。经历过刚学 Java 时的迷茫，第一次写出 Java 程序时的激动，第一次写出一个 Web 系统的醍醐灌顶，一直到接触 Java 更底层的东西，总的来说对 Java 有了系统性的认识，对后端技术体系有了宏观的感受。这期间，笔者用过各种各样的编程语言，尝试过各种开源软件，挖过各种坑，也填过各种坑。针对后端技术来说，笔者认为自己的这些知识体系，还是有一定价值的。

此外，还记得当笔者毕业后进入第一家公司时，入职培训的课程虽然不难，但确实有种恍然大悟的感觉。业界的最佳实践和自己在学校里学到的、使用到的知识，差别还是非常大的。直到后来加入现在的这家公司，给新老员工做过一系列后端技术的培训课程，在校招的笔试和面试过程中深刻体会到学校与业界脱节之严重，在平时的社招中遇到很多对后端技术缺乏系统性认识、技能点不足的工程师，并且也经常被人问起如何学习 Java 后端技术，于是就打算将目前后端工程师一些比较主流、前沿的技术以及实际工作中会用到的技能串联起来，给刚上大学以后打算以 Java 后端为职业的学生、刚毕业入职的应届生以及初学者们一些入门的指引，使其少走弯路。另外也希望给一些有经验的工程师提供一个参考手册，将零散的知识点串起来，减少在解决某些实际问题时无头绪搜索带来的时间成本，同时也是对自己的一个阶段性总结和查漏补缺。需要注意的一点是，像数据结构、计算机网络等计算机科学基础知识以及 Java SE 的基本用法，笔者认为是从事程序开发工作的 Java 工程师应该必备的知识，因此并不包括在内。

本书会针对 Java 后端开发工作中经常用到的关键技能点做阐述，会尽量覆盖实际工作中需要的所有技能。但由于很多技能并非一两个章节就能完整讲述，因此本书仅做一些实践性的经验总结和阐述，更加详细和深入的学习则需要参考专门的书籍或者官方文档。

本书的大部分内容都来自笔者的博客以及平时工作、学习中的一些自我总结和笔记，记录了笔者进入这个行业以来的一些经验教训和思考。

## 面向读者

- 未入门或者刚入门的 Java 工程师

包括未来以 Java 后端开发为职业方向的在校学生、刚毕业入职的 Java 工程师以及未形成知识体系的 Java 工程师。这类读者通过阅读本书能够对 Java 工程师的必备技能有一个全局认识，逐步形成自己的 Java 技术体系。

- 有经验的 Java 工程师

有经验的 Java 工程师可以通过本书查漏补缺，巩固自己的开发技能，进一步完善自身的 Java 技术体系。

- 对 Java 后端开发感兴趣的非 Java 工程师

非 Java 工程师可以通过本书了解 Java 工程师的技能体系，尤其对于其他语言的后端工程师来说，本书的很多内容也是通用的，并不局限于 Java 开发。

## 内容概览

- 第 1 章 后端技术导言

本章主要从总体上描述后端技术的概念、组成、作用、需要的知识点，并给出了学习后端技术的建议。

- 第 2 章 Java 项目与工程化

本章主要讲述 Java 项目与工程化需要掌握的软件、技能等。

- 第 3 章 开发框架

本章主要讲述 Java 后端开发中的一些主流框架的使用方法。

- 第 4 章 Spring

本章主要讲述 Spring 核心、数据操作以及一些常用组件的使用。

- 第5章 数据存储

本章主要讲述 Java 应用中数据存储上使用的一些软件、服务等。

- 第6章 数据通信

本章主要讲述 Java 应用中数据传输、通信上使用的一些软件、服务等。

- 第7章 Java 编程进阶

本章主要介绍一些 Java 开发中的高级特性以及在 Java 开发中非常流行的类库。

- 第8章 性能调优

本章主要讲述如何对 Java 应用的性能进行分析和调优，并给出了开发建议。

- 第9章 安全技术

本章主要对 Java 开发中常用的加密技术、HTTPS 以及防范各种攻击的方案做了阐述。

## 参考资料

在写作本书的过程以及平时的工作中，笔者阅读、参考过很多书籍，以下是其中具有代表性的一些书籍。对于本书讲述不够深入的地方，可以参考这些书籍进一步学习。

- 《Effective Java（第2版）》：此书讲解了 Java 的一些高级特性和技巧。
- 《Java 并发编程实战》：此书是并发编程经典书籍，涵盖了并发编程的各种知识点以及相关理论知识。
- 《七周七并发模型》：此书讲述了主流的 7 种并发编程模式。
- 《深入理解 Java 虚拟机：JVM 高级特性与最佳实践（第2版）》：此书讲解了 JVM 的内存、GC、字节码、编译器等高级特性和优化实践。
- 《高性能 MySQL（第3版）》：此书讲述了 MySQL 各种优化技巧，并结合原理给予讲解。
- 《Redis 开发与运维》：此书在原理层面对于 Redis 的使用、优化做了详尽的描述。
- 《深入理解 Elasticsearch（原书第2版）》：此书讲述了对 Elasticsearch 的使用、原理和优化技巧。
- 《Java 性能权威指南》：此书是 Java 性能调优的权威书籍，几乎涵盖了 Java 调优的方方面面。

- 《构建高性能 Web 站点（修订版）》：此书从各种案例出发，讲解了高性能 Web 站点需要的各种优化技巧、实践经验等。
- 《白帽子讲 Web 安全》此书基本涵盖了 Web 安全技术的方方面面，包括客户端安全、服务器端安全等。

虽然以上书籍都是非常实用的参考资料，但就笔者自己来看，更为推崇的则是直接通过相关技术的官方文档来学习，这样既能够锻炼自己的英文阅读能力，又能够直面相关技术的第一手文档，避免了在看相关书籍时被一些偶然的纰漏所误导。

此外，上面的《Effective Java（第2版）》和《Java 并发编程实战》这两本书都是基于 Java 的旧版本来写作的，但是里面介绍的很多内容并不过时，尤其是 JDK 底层源码、设计理论、优化思想等仍然适用于现在的 Java 开发。

## 勘误和支持

在本书的写作过程中，笔者一直是战战兢兢的，一直害怕写成那种侃侃而谈却没有实质内容的东西或者传递给读者一些误导信息，因此对于每一个知识点，都是在查阅官方文档以及其他权威资料并经过自己深入思考之后才敢落笔的。但由于笔者知识能力有限，难免有错误和疏漏，希望得到各位读者的理解和指正。

如果在阅读本书的过程中发现错误，请提交到网址 <https://github.com/superhj1987/pragmatic-java-engineer/issues> 或博文视点官网本书页面。

同时，请随时注意勘误信息的发布：<https://github.com/superhj1987/pragmatic-java-engineer/wiki/Mistakes>。

## 致谢

由于工作以及个人身体等方面的原因，中间数次延期，历时一年多才完成本书。因此首先要特别感谢笔者的父母和妻子，在笔者写作本书的过程中给予了非常大的后勤支持和鼓励，让笔者能够专心地完成写作。

同时要感谢中华万年历的同事们在平时的工作中给了笔者很多启发和思路，感谢公司的设计总监张喜亮抽出时间帮助修饰了一些图片，尤其要感谢 CEO 孙建在本书写作过程中给予笔者了充分的信任和支持。

还要感谢笔者的前同事，也是笔者刚毕业时的工作导师尧飘海，他在百忙之中审阅了本书并给本书写序；也要感谢前同事张小川、阙杭宁和好友秦绪震、饶洵，他们抽出了宝贵的业余时间校对了本书并给出了很有价值的建议。

最后，感谢电子工业出版社永恒的侠少，他找到我出版本书，并允许我一次次延期。也感谢付睿编辑的辛苦校对和修改，让本书得以顺利出版。

也把本书献给我刚出生的女儿——依依。

## 联系方式

邮箱: [superhj1987@126.com](mailto:superhj1987@126.com)

博客: <http://rowkey.me>

微博: <http://weibo.com/superhj1987>

## 读者服务

轻松注册成为博文视点社区用户（[www.broadview.com.cn](http://www.broadview.com.cn)），您即可享受以下服务。

- **下载资源：**本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口: <http://www.broadview.com.cn/33501>



# 目录

第 1 章 后端技术导言 .....	1
1.1 后端基础设施 .....	2
1.1.1 请求统一入口——API 网关 .....	3
1.1.2 业务应用和后端基础框架 .....	4
1.1.3 缓存、数据库、搜索引擎、消息队列 .....	5
1.1.4 文件存储 .....	6
1.1.5 统一认证中心 .....	6
1.1.6 单点登录系统 .....	7
1.1.7 统一配置中心 .....	7
1.1.8 服务治理框架 .....	8
1.1.9 统一调度中心 .....	9
1.1.10 统一日志服务 .....	10
1.1.11 数据基础设施 .....	10
1.1.12 故障监控 .....	13
1.2 Java 后端技术概览 .....	14
1.2.1 软件开发的核心原则 .....	15
1.2.2 软件开发的过程管理 .....	15
1.2.3 日常开发常用工具 .....	16
1.2.4 应用的运行环境 .....	16
1.2.5 常用第三方服务 .....	17
1.2.6 计算机基础科学知识 .....	18

1.2.7	数据处理相关技能 .....	19
1.2.8	Java 编程知识 .....	21
1.2.9	系统架构演化 .....	22
1.2.10	典型的部署架构 .....	23
1.3	如何学习后端技术 .....	24
1.3.1	扎实的计算机基础知识 .....	25
1.3.2	知其然更要知其所以然 .....	26
1.3.3	动手实践 .....	26
1.3.4	频繁练习 .....	26
1.3.5	持续学习 .....	27
1.3.6	自我总结 .....	27
1.3.7	如何学习一门新技术 .....	28
1.3.8	小结 .....	29
<b>第 2 章</b>	<b>Java 项目与工程化 .....</b>	<b>30</b>
2.1	项目构建 .....	31
2.1.1	传统构建工具——Ant .....	31
2.1.2	主流构建工具——Maven .....	34
2.1.3	新兴构建工具——Gradle .....	43
2.2	代码版本控制 .....	46
2.2.1	集中式代码版本管理——SVN .....	46
2.2.2	分布式代码版本管理——Git .....	49
2.2.3	提交日志的规范 .....	55
2.3	代码质量保证 .....	58
2.3.1	使用单元测试保证代码质量 .....	59
2.3.2	衡量单元测试的标准 .....	66
2.3.3	开发规范与建议 .....	67
<b>第 3 章</b>	<b>开发框架 .....</b>	<b>70</b>
3.1	依赖注入 .....	71
3.1.1	JSR-330 依赖注入规范 .....	73
3.1.2	Guice .....	75

3.1.3	PicoContainer .....	76
3.1.4	Dagger.....	76
3.1.5	Spring Framework.....	77
3.1.6	循环依赖问题.....	79
3.2	对象关系映射 .....	79
3.2.1	表元数据的映射.....	80
3.2.2	CRUD 以及属性的查询 .....	82
3.2.3	查询缓存的使用.....	83
3.2.4	结果的映射 .....	84
3.2.5	规范 SQL 书写的语句构建器 .....	84
3.2.6	使用提示 .....	85
3.3	日志 .....	86
3.3.1	JDK Logging .....	87
3.3.2	Log4j .....	88
3.3.3	Log4j2.....	90
3.3.4	Logback.....	93
3.3.5	统一日志 API 的门面框架.....	95
3.3.6	统一日志框架的使用 .....	98
3.4	Web MVC .....	99
3.4.1	为什么是 Spring MVC .....	99
3.4.2	Spring MVC 的请求处理流程.....	100
3.4.3	典型的配置方式.....	102
3.4.4	无 XML 的配置方式 .....	105
3.4.5	对 MVC 应用做单元测试 .....	106
3.4.6	验证 Web 请求的参数.....	107
3.4.7	使用异步 Servlet.....	110
3.4.8	使用提示 .....	112
第 4 章	Spring.....	115
4.1	Spring 核心组件 .....	117
4.1.1	Spring 的双亲上下文机制 .....	118
4.1.2	Spring 中的事件机制 .....	119
4.1.3	Bean 的初始化和销毁.....	120



4.1.4	Bean 的动态构造 .....	122
4.1.5	注入集合、枚举、类的静态字段 .....	124
4.1.6	面向方面编程——AOP .....	125
4.1.7	进阶 XML 的配置 .....	130
4.1.8	无 XML 的配置方式 .....	133
4.2	Spring 数据操作框架 .....	135
4.2.1	Spring JDBC .....	135
4.2.2	Spring Data Redis .....	136
4.2.3	Spring Data MongoDB .....	138
4.3	Spring Boot .....	140
4.3.1	Spring Boot 使用示例 .....	140
4.3.2	Spring Boot 的运行原理 .....	141
4.3.3	Spring Boot 的组成模块 .....	143
4.3.4	小结 .....	144
4.4	Spring 常用组件 .....	144
4.4.1	表达式引擎——Spring Expression Language .....	144
4.4.2	远程过程访问的支持——Spring Remoting .....	145
4.4.3	Spring 与 JMX 的集成 .....	146
4.4.4	定时任务的支持——Spring Quartz .....	147
4.4.5	跨域请求的支持——Spring CORS .....	148
4.5	总结 .....	149
<b>第 5 章</b>	<b>数据存储 .....</b>	<b>151</b>
5.1	关系型数据库——MySQL .....	152
5.1.1	存储引擎 .....	152
5.1.2	字符集和校对规则 .....	153
5.1.3	索引的使用 .....	154
5.1.4	查询缓存的使用 .....	158
5.1.5	数据同步中的 Binlog .....	159
5.1.6	事务机制 .....	159
5.1.7	大表优化 .....	163
5.1.8	高可用支持 .....	164
5.1.9	使用提示 .....	166

5.2	非关系型数据库 .....	169
5.2.1	KV 数据库 .....	170
5.2.2	文档数据库——MongoDB .....	171
5.2.3	列数据库——HBase .....	181
5.3	缓存 .....	185
5.3.1	本地缓存 .....	186
5.3.2	分布式缓存——Redis .....	188
5.3.3	缓存设计的典型方案 .....	195
5.4	搜索引擎——Elasticsearch .....	196
5.4.1	开源全文检索库——Apache Lucene .....	197
5.4.2	关键概念 .....	198
5.4.3	查询的优化 .....	199
5.4.4	内存的使用优化 .....	201
5.4.5	开源日志管理方案——ELK .....	202
<b>第 6 章</b>	<b>数据通信 .....</b>	<b>204</b>
6.1	RESTful 架构风格 .....	204
6.1.1	支持的操作 .....	205
6.1.2	返回码 .....	206
6.1.3	资源概念 .....	207
6.1.4	数据的安全保障 .....	208
6.1.5	请求的限流 .....	210
6.1.6	超文本 API .....	211
6.1.7	编写文档 .....	211
6.1.8	RESTful API 实现 .....	211
6.2	远程过程调用——RPC .....	212
6.2.1	JDK 自带的 RPC——RMI .....	213
6.2.2	Hessian .....	213
6.2.3	Thrift .....	214
6.2.4	Dubbo .....	216
6.2.5	数据的序列化机制 .....	217
6.2.6	使用提示 .....	222

6.3 消息中间件.....	222
6.3.1 简单消息中间件——ActiveMQ .....	224
6.3.2 通用消息中间件——RabbitMQ .....	225
6.3.3 日志消息中间件——Kafka .....	230
6.3.4 本地消息队列.....	237
 第 7 章 Java 编程进阶.....	241
7.1 Java 内存管理.....	242
7.1.1 JVM 虚拟机内存 .....	242
7.1.2 垃圾回收理论.....	245
7.1.3 常用垃圾回收器.....	251
7.2 Java 网络编程.....	255
7.2.1 常见网络 I/O 模型.....	256
7.2.2 Java 网络编程模型.....	261
7.3 Java 并发编程.....	263
7.3.1 并发原理 .....	263
7.3.2 并发思路 .....	268
7.3.3 并发工具 .....	270
7.3.4 并发编程建议.....	273
7.4 Java 开发利器.....	273
7.4.1 Apache 工具库——Apache Commons.....	274
7.4.2 Google 工具库——Guava .....	284
7.4.3 最好用的时间库——Joda Time .....	288
7.4.4 高效 JSON 处理库——FastJson .....	289
7.4.5 高效 Bean 映射框架——Orika.....	290
7.5 Java 新版本的特性 .....	291
7.5.1 Java 7.....	291
7.5.2 Java 8.....	293
7.5.3 Java 9.....	300
7.6 总结 .....	303

第 8 章 性能调优.....	304
8.1 调优准备.....	305
8.1.1 HotSpot 虚拟机体系结构.....	306
8.1.2 操作系统的性能调优.....	307
8.1.3 系统常用诊断工具.....	310
8.1.4 JDK 常用诊断工具.....	313
8.2 性能分析.....	315
8.2.1 CPU 分析.....	315
8.2.2 内存分析.....	317
8.2.3 I/O 分析.....	318
8.2.4 其他分析工具.....	320
8.3 性能调优.....	325
8.3.1 CPU 调优.....	325
8.3.2 内存调优.....	325
8.3.3 I/O 调优.....	328
8.3.4 其他优化建议.....	329
8.3.5 JVM 参数配置.....	329
8.3.6 JVM 性能增强.....	331
第 9 章 安全技术.....	333
9.1 Java 加密.....	333
9.1.1 单向加密算法.....	334
9.1.2 对称加密算法.....	335
9.1.3 非对称加密算法.....	338
9.2 安全 HTTP——HTTPS.....	341
9.2.1 安全协议——SSL/TLS.....	342
9.2.2 证书中心——CA.....	343
9.2.3 请求交互过程.....	344
9.2.4 性能优化.....	345
9.3 Web 安全.....	346
9.3.1 跨站点脚本攻击.....	347
9.3.2 跨站点请求伪造.....	347

9.3.3	SQL 注入攻击 .....	348
9.3.4	基于约束条件的 SQL 攻击 .....	349
9.3.5	分布式拒绝服务攻击——DDOS .....	350
9.3.6	会话固定攻击——Session fixation .....	351
附录 A	代码构建常用命令 .....	353
附录 B	Git 常用命令 .....	356
附录 C	MySQL 常用命令 .....	367
附录 D	MongoDB 常用命令 .....	373
附录 E	Java 调优常用命令 .....	379

# 第 1 章

## 后端技术导言

淘宝双 11、京东 618，大家疯狂抢购各种降价商品，不断确认提交订单，商家不断地发出快递；使用今日头条，无数你感兴趣的内容出现在你的面前，让你能够实时掌握你最关心的消息；使用滴滴，你快速又便宜地打到了车，到达了目的地；使用大众点评，你迅速找到了周围最受欢迎的饭店，吃到了可口的饭菜；使用中华万年历，你实时快速地知道了明天的天气预报，是否是黄道吉日，做好了一天的日程规划，诸如此类。日常生活中这些 APP 给大家的生活带来了翻天覆地的改变，老百姓的衣食住行也越来越依赖于这些 APP。在这些或绚丽或实用的 APP 后面到底是什么技术在支撑着它们的运行呢？

从大的范围来讲，支撑这些 APP 的主要有前端、后端这两种技术。前端指的是用户能够直接感知到的那些东西，包括 Web 前端和客户端技术；而后端技术是相对于前端技术而言的，是藏在网络后面支撑网页、APP、应用软件运行的设施、环境、服务等。通过这两种技术的结合，就产生了一个稳定的互联网软件，可以给用户提供持续稳定的服务。其中后端技术通过几台甚至成千上万台服务器给前端提供高质量的服务，扮演了一个基础设施、运转轴心的角色。

相比前端技术的种类繁多以及不可预测性（比如诺基亚的 Symbian 在火爆了  $n$  年以后瞬间没落，相应的编程技术也随着失去了用武之地），后端技术还是比较稳定的，即使新的技术层出不穷，旧的应用比较广泛的技术也不会突然就走下历史舞台。但是后端技术的涵盖范围实在太广，Web、大数据、高并发、数据库、数据挖掘、机器学习……任何一个技能点单拎出来都需要花费极大的精力才能做到融会贯通。因此这并非一个人能够全部掌握的，即使是很多所谓的全栈工程师，其实也就掌握了系统前后端的基本技能而已，要说做到精通，则没几个人敢妄言。

由于前端技术和后端技术与用户接触的层次不一样，很多关注点也是不一样的。对于后端技术来讲，一般情况下需要关注的是以下几个指标。

- **可用率**：能够提供正常服务的时间占线上运行时间的百分比。这是后端服务中一个很关键的指标，很多应用都需要达到 99.9% 以上。这里需要注意的是，系统的可用性经常是用响应时间作为衡量指标的，关注的是系统提供正常服务的效率；与响应时间类似，吞吐量是衡量系统可用性的另一个指标，指的是系统一段时间内的处理能力。
- **稳定性**：也叫作鲁棒性、健壮性，即服务在异常和危险情况下保持稳定的能力。
- **容错性**：在服务出现错误或者异常的时候，能够继续提供一定服务的能力，主要强调的是容许误差、故障的能力。
- **扩展性**：主要指服务的动态扩展能力，即通过扩展（而非修改）现有系统的能力来满足需求的能力。
- **可维护性**：指的是修正服务错误、修改服务功能的能力。
- **安全性**：保障系统以及用户数据安全的能力，包括保障系统不被非法入侵、用户数据不被泄露等。

除此之外很多书籍和资料上还会有可靠性、可用性等说法，其本质上是以上指标的另一种说法而已。一般来说后端服务只要满足了高可用、可容错、可扩展、可维护、稳定性、安全性即可。后端技术也基本是围绕着这几个目标来进行的。

综上，由于后端技术覆盖范围太广以及本人知识所限，本书所说的后端技术主要指一些 Java 后端工程师能够胜任开发工作应该必备的一些技能。

### 1.1 后端基础设施

使用 Java 后端技术的目的就是构建业务应用，为用户提供在线或者离线服务。因此，一个业务应用需要哪些技术、依赖哪些基础设施就决定了需要掌握的后端技术有哪些。纵观整个互联网技术体系再结合公司的目前状况，笔者认为必不可少或者非常关键的后端基础技术 / 设施如图 1-1 所示。

这里的后端基础设施主要指的是应用软件在线上稳定运行需要依赖的关键组件或者服务。开发或者搭建好以上的后端基础设施，一般情况下是能够支撑很长一段时间内的业务的。此外，对于一个完整的架构来说，还有很多应用感知不到的系统基础服务，如负载均衡、自动化部署、系统安全等，并没有包含在本章的描述范围内。

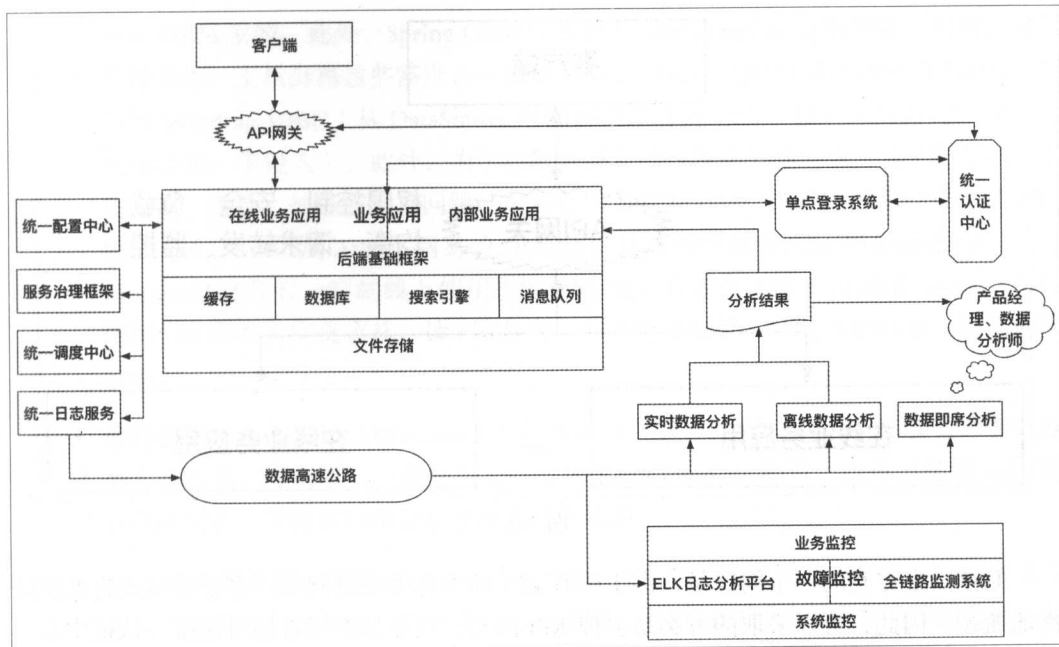


图 1-1

### 1.1.1 请求统一入口——API 网关

在移动 APP 的开发过程中，通常后端提供的接口需要以下功能的支持。

- 负载均衡。
- API 访问权限控制。
- 用户鉴权。

一般的做法是，使用 Nginx 做负载均衡，然后在每个业务应用里做 API 接口的访问权限控制和用户鉴权，更优化一点的方式则是把后两者做成公共类库供所有业务调用。但从总体上看，这 3 种特性都属于业务的公共需求，更可取的方式则是集成到一起作为一个服务，这样既可以动态地修改权限控制和鉴权机制，也可以减少每个业务集成这些机制的成本。这种服务就是 API 网关，可以选择自己实现，也可以使用开源软件实现，如 Kong。一般 API 网关的架构如图 1-2 所示。



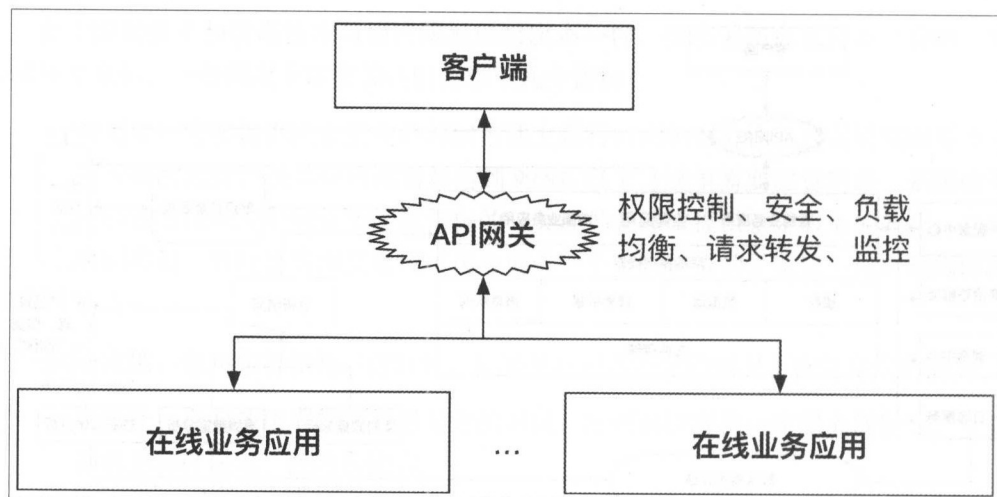


图 1-2

但是以上方案的一个问题是，由于所有 API 请求都要经过网关，它很容易成为系统的性能瓶颈。因此，可以采取的方案是去掉 API 网关，让业务应用直接对接统一认证中心，在基础框架层面保证每个 API 调用都需要先通过统一认证中心的认证，这里可以采取缓存认证结果的方式避免对统一认证中心产生过大的请求压力。

### 1.1.2 业务应用和后端基础框架

业务应用分为在线业务应用和内部业务应用，分别介绍如下。

- **在线业务应用**：直接面向互联网用户的应用、接口等，典型的特点就是请求量大、高并发、对故障的容忍度低。
- **内部业务应用**：主要面向公司内部用户的应用。比如，内部数据管理平台、广告投放平台等。相比在线业务应用，其特点是数据保密性高、压力小、并发量小、允许故障的发生。

业务应用是基于后端的基础框架进行开发的，针对 Java 后端来说，有以下几个框架。

- **MVC 框架**：是统一开发流程、提高开发效率、屏蔽一些关键细节的 Web/ 后端框架。典型的 MVC 框架如 Spring MVC、Jersey、国人开发的 JFinal 以及阿里的 WebX。
- **IoC 框架**：可实现依赖注入 / 控制反转的框架。Java 中最流行的 Spring 框架的核心就是 IoC 功能。
- **ORM 框架**：是能够屏蔽底层数据库细节、提供统一的数据访问接口的数据库操作框架，另外也能够支持客户端主从、分库、分表等分布式特性。MyBatis 是目前最流

行的 ORM 框架。此外, Spring ORM 中提供的 JdbcTemplate 也很不错。当然, 对于分库分表、主从分离这些需求, 一般需要自己实现, 开源的则有阿里的 TDDL、当当的 Sharding-JDBC (从 DataSource 层面解决了分库分表、读 / 写分离的问题, 对应用透明、零侵入)。此外, 为了在服务层面统一解决分库分表、读 / 写分离、主备切换、缓存、故障恢复等问题, 很多公司都开发了自己的数据库中间件, 比如阿里的 Cobar、360 的 Atlas、网易的 DDB 等; 开源的则有 MyCat 和 Kingshard, 其中 Kingshard 已经有一定的线上使用规模。MySQL 官方也提供了 MySQL Proxy, 可以使用 Lua 脚本自定义主从、读 / 写分离、分区这些逻辑, 但其性能较差, 目前使用较少。

- **缓存框架:** 对 Redis、Memcached 这些缓存软件操作的统一封装, 能够支持客户端分布式方案、主从等。一般使用 Spring 的 RedisTemplate 即可, 也可以使用 Jedis 做自己的封装, 支持客户端分布式方案、主从等。
- **Java EE 应用性能检测框架:** 对于线上的 Java EE 应用, 需要有一个统一的框架集成到每一个业务中检测每一个请求、方法调用、JDBC 连接、Redis 连接等的耗时、状态等。Jwebap 是一个可以使用的性能检测工具, 但由于其已经很多年没有更新, 有可能的话建议基于此项目做二次开发。

一般来说, 以上几个框架即可完成一个后端应用的雏形。

### 1.1.3 缓存、数据库、搜索引擎、消息队列

缓存、数据库、搜索引擎、消息队列这 4 者都是应用依赖的后端基础服务, 它们的性能直接影响到应用的整体性能, 有时候你代码写得再好也可能因为这些服务导致应用性能无法提升上去。

- **缓存:** 缓存通常被用来解决热点数据的访问问题, 是提高数据查询性能的强大武器。在高并发的后端应用中, 将数据持久层的数据加载到缓存中, 能够隔离高并发请求与后端数据库, 避免数据库被大量请求击垮。目前除了内存中的本地缓存, 比较普遍使用的缓存软件有 Memcached 和 Redis, 其中 Redis 已经成为最主流的缓存软件。
- **数据库:** 数据库可以说是后端应用最基本的基础设施。基本上绝大多数业务数据都是持久化存储在数据库中的。主流的数据库包括传统的关系型数据库 (MySQL、PostgreSQL) 以及最近几年开始流行的 NoSQL (MongoDB、HBase)。其中 HBase 是用于大数据领域的列数据库, 受限于其查询性能, 一般并不用于做业务数据库。

- **搜索引擎**：搜索引擎是针对全文检索以及数据各种维度查询设计的软件。目前用得比较多的开源软件是 Solr 和 Elasticsearch，它们都是基于 Lucene 来实现的，不同之处主要在于 Term Index 的存储、分布式架构的支持等。Elasticsearch 由于对集群的良好支持以及高性能的实现，已经逐渐成为搜索引擎的主流开源方案。
- **消息队列**：数据传输的一种方式就是通过消息队列。目前用得比较普遍的消息队列包括为日志设计的 Kafka 以及重事务的 RabbitMQ 等。在对消息丢失不是特别敏感且并不要求消息事务的场景下，选择 Kafka 能够获得更高的性能；否则，RabbitMQ 则是更好的选择。

### 1.1.4 文件存储

不管是业务应用、依赖的后端服务还是其他的各种服务，最终都要依赖于底层文件存储。通常来说，文件存储需要满足的特性有：可靠性、容灾性、稳定性，既要保证存储的数据不轻易丢失，即使发生故障也能够有回滚方案，也要保证高可用。在底层可以采用传统的 RAID 作为解决方案，再上一层，目前 Hadoop 的 HDFS 则是最普遍的分布式文件存储方案，当然还有 NFS、Samba 这种共享文件系统也提供了简单的分布式存储的特性。

此外，如果文件存储确实成为了应用的瓶颈，或者必须提高文件存储的性能从而提升整个系统的性能，那么最直接和简单的做法就是抛弃传统机械硬盘，用 SSD 硬盘替代。像现在很多公司在解决业务性能问题的时候，最终的关键点往往就是 SSD。这也是用钱换取时间和人力成本最直接和最有效的方式。在数据库部分描述的 SSDB，就是对 LevelDB 封装之后，利用 SSD 硬盘特性的一种高性能 KV 数据库。

至于 HDFS，如果要使用上面的数据，需要通过 Hadoop 来实现。类似 XX on YARN 的一些技术就是将非 Hadoop 技术跑在 HDFS 上的解决方案。

### 1.1.5 统一认证中心

统一认证中心，主要是对 APP 用户、内部用户、APP 等的认证服务，包括

- 用户的注册、登录验证、Token 鉴权。
- 内部信息系统用户的管理和登录鉴权。
- APP 的管理，包括 APP 的 secret 生成、APP 信息的验证（如验证接口签名）等。

之所以需要统一认证中心，就是为了能够集中对所有 APP 都会用到的信息进行管理，也给所有应用提供统一的认证服务。尤其是在有很多业务需要共享用户数据的时候，构建

一个统一认证中心是非常必要的。此外，通过统一认证中心构建移动 APP 的单点登录也是水到渠成的事情：模仿 Web 的机制，将认证后的信息加密存储到本地存储中，供多个 APP 使用。

### 1.1.6 单点登录系统

目前很多大的在线 Web 网站都是有单点登录系统的，通俗地说，就是只需要一次用户登录，就能够进入多个业务应用（权限可以不相同），非常方便用户操作。而在移动互联网公司中，内部的各种管理、信息系统，甚至外部应用同样也需要单点登录系统。

目前，比较成熟的、用得最多的单点登录系统应该是耶鲁大学开源的 CAS，可以基于 <https://github.com/apereo/cas/tree/master/cas-server-webapp> 来定制开发。

基本上，单点登录的原理都类似于图 1-3。

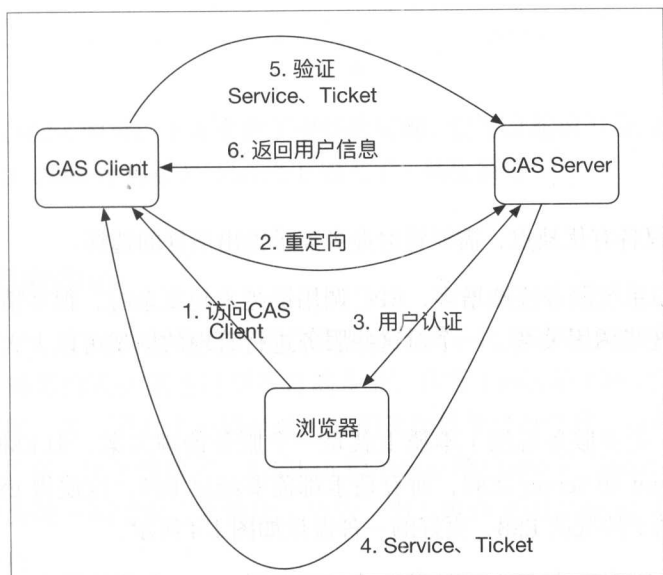


图 1-3

### 1.1.7 统一配置中心

在 Java 后端应用中，一种读 / 写配置比较通用的方式就是将配置文件写在 Properties、YAML、HCON 等文件中，修改的时候只需要更新文件重新部署，可以做到不牵扯代码层面改动的目的。统一配置中心，则是基于这种方式之上的统一对所有业务或者基础后端服务的相关配置文件进行管理的服务，具有以下特性。

- 能够在线动态修改配置文件并生效。
- 配置文件可以区分环境（开发、测试、生产等）。
- 在 Java 中可以通过注解、XML 配置的方式引入相关配置。

百度开源的 Disconf 是一个可以在生产环境中使用的方案，也可以根据自己的需求开发自己的配置中心，一般选择 ZooKeeper 作为配置存储。

### 1.1.8 服务治理框架

对于外部 API 调用或者客户端对后端 API 的访问，可以使用 HTTP 协议或者 RESTful（当然也可以直接通过最原始的 Socket 来调用）。但对于内部服务间的调用，一般都是通过 RPC 机制来进行的。目前主流的 RPC 协议有：

- RMI。
- Hessian。
- Thrift。
- Dubbo。

这些 RPC 协议各有优缺点，需要针对业务需求做出最好的选择。

这样，当你的系统服务逐渐增多，RPC 调用链越来越复杂时，很多情况下需要不停地更新文档来维护这些调用关系。一个对这些服务进行管理的框架可以大大减少因此带来的烦琐的人力工作。

传统的 ESB（企业服务总线）本质上就是一个服务治理方案，但 ESB 作为一种 Proxy 的角色存在于 Client 和 Server 之间，所有请求都需要经过 ESB，这使得 ESB 很容易成为性能瓶颈。因此，基于传统的 ESB，更好的一种设计如图 1-4 所示。

图 1-4 中以配置中心为枢纽，调用关系只存在于 Client 和提供服务的 Server 之间，这就避免了传统 ESB 的性能瓶颈问题。对于这种设计，ESB 应该支持的特性如下。

- 服务提供方的注册、管理。
- 服务消费者的注册、管理。
- 服务的版本管理、负载均衡、流量控制、服务降级、资源隔离。
- 服务的容错、熔断。

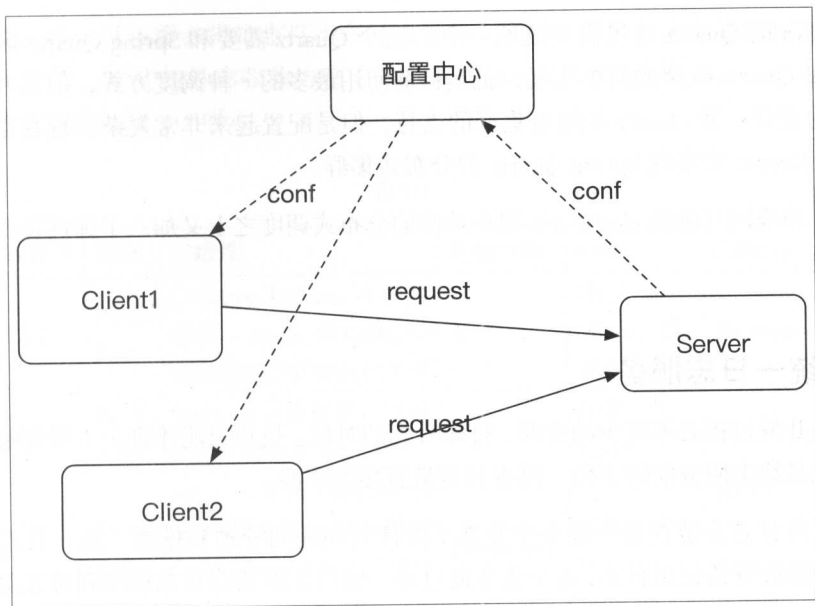


图 1-4

阿里开源的 Dubbo 则对以上方案做了很好的实现，也是目前很多公司都在使用的方案；当当网的扩展项目 Dubbox 则在 Dubbo 之上加入了一些新特性。

### 1.1.9 统一调度中心

在很多业务中，定时调度是一个非常普遍的场景，比如定时去抓取数据、定时刷新订单的状态等。通常的做法就是针对各自的业务，依赖 Linux 的 Cron 机制或者 Java 中的 Quartz 来实现调度。统一调度中心则是对所有的调度任务进行管理，这样能够统一对调度集群进行调优、扩展、任务管理等。Azkaban 和 Yahoo 的 Oozie 是 Hadoop 的流式工作管理引擎，也可以作为统一调度中心来使用。当然，你也可以使用 Cron 或者 Quartz 来实现自己的统一调度中心。

- 根据 Cron 表达式调度任务。
- 动态修改、停止、删除任务。
- 支持任务分片执行。
- 支持任务工作流，比如一个任务完成之后再执行下一个任务。
- 任务支持脚本、代码、URL 等多种形式。
- 任务执行的日志记录、故障报警。

对于 Java 的 Quartz 这里需要说明一下：这个 Quartz 需要和 Spring Quartz 区分，后者是 Spring 对 Quartz 框架的简单实现，也是目前使用最多的一种调度方式，但其并没有做高可用集群的支持。而 Quartz 虽然有集群的支持，但是配置起来非常复杂。现在很多方案都是使用 ZooKeeper 来实现 Spring Quartz 的分布式集群。

此外，当当网开源的 elastic-job 则在基础的分布式调度之上又加入了弹性资源利用等更强大的功能。

### 1.1.10 统一日志服务

日志是开发过程必不可少的东西。打印日志的时机、技巧很能体现出工程师编码水平。毕竟，日志是线上服务能够定位、排查异常最直接的来源。

通常，将日志分散在各个业务中非常不方便对问题的管理和排查。统一日志服务则使用单独的日志服务器记录日志，各个业务通过统一的日志框架将日志输出到日志服务器上。

可以通过实现 Log4j 或者 LogBack 的 appender 来实现统一日志框架，然后通过 RPC 调用将日志打印到日志服务器上。

### 1.1.11 数据基础设施

数据是最近几年非常火的一个领域。从《精益数据分析》到《增长黑客》，都是在强调数据的非凡作用。很多公司也都在通过数据推动产品设计、市场运营、研发等。这里需要说明的一点是，只有当你的数据规模真的到了单机无法处理的规模时才应该上大数据相关技术，千万不要为了大数据而大数据。很多情况下使用单机程序+MySQL 就能解决的问题，如盲目地上 Hadoop 则会既浪费时间又浪费人力。

这里需要补充一点的是，对于很多公司，尤其是离线业务没有那么密集的公司，在很多情况下大数据集群的资源是被浪费的。因此诞生了 XX on YARN 一系列技术，让非 Hadoop 系的技术可以利用大数据集群的资源，这样能够大大提高资源的利用率，如 Docker on YARN。

### 数据高速公路

对于上面介绍的统一日志服务，其输出的日志最终会变成数据，到数据高速公路上供后续的数据处理程序消费。这中间的过程包括日志的收集和传输。

- **收集：**统一日志服务将日志打印在日志服务上之后，需要日志收集机制将其集中起来。目前，常见的日志收集方案有 Scribe、Chukwa、Kakfa 和 Flume。几种日志收集方案对比如表 1-1 所示。

表 1-1

	实现语言	框架	容错性	负载均衡	Agent	Collector	Store
Scribe	C/C++	Push/ Push	Collector 和 Store 间有容错性，Agent 和 Collector 间的容错性需要自己实现	无	Thrift Client，需要自己实现	Thrift Server	支持 HDFS
Chukwa	Java	Push/ Push	Agent 记录偏移量，出现故障可继续发送	无	自带一些 Agent	—	支持 HDFS
Kafka	Scala	Push/ Pull	Agent 可通过 Collector 自动识别机制获取可用的 Collector，Store 自己保存 offset	ZooKeeper	需要自己实现	使用了 Senfile，Zero-Copy 提高性能	支持 HDFS
Flume	Java	Push/ Push	Agent、Collector 及 Store 之间均有容错机制，3 种级别的可靠保证	ZooKeeper	提供了丰富的 Agent	提供了很多 Collector	支持 HDFS

此外，Logstash 也是一个可以选择的日志收集方案，不同于以上几种方案的是，它更倾向于数据的预处理，且配置简单、清晰，经常以 ELK（Elasticsearch + Logstash + Kibana）的架构用于运维场景中。

- **传输：**通过消息队列将数据传输到数据处理服务中。对于日志来说，通常选择 Kafka 这个消息队列即可。

此外，这里还有一个关键之处，就是数据库和数据仓库间的数据同步问题，要有将需要分析的数据从数据库同步到诸如 Hive 这种数据仓库时使用的方案。可以使用 Apache Sqoop 进行基于时间戳的数据同步，此外，阿里开源的 Canal 实现了基于 Binlog 的增量同步，更加适合通用的同步场景，但是基于 Canal 还需要做不少的业务开发工作。

离线数据分析

离线数据分析是可以有延迟的，一般针对的是非实时需求的数据分析工作，产生的也是延迟 1 天的报表。目前最常用的离线数据分析技术除了 Hadoop 还有 Spark。相比 Hadoop，Spark 在性能上有很大优势，当然对硬件资源要求也高。

对于 Hadoop，传统的 MR 编写很复杂，也不利于维护，可以选择使用 Hive 来用 SQL 替代编写 MR。而对于 Spark，也有类似 Hive 的 Spark SQL。



此外，对于离线数据分析，还有一个很关键的问题就是数据倾斜。数据倾斜指的是 Region 数据分布不均，造成有些节点负载很低，而有些却负载很高，从而影响整体性能。处理好数据倾斜问题对于数据处理是很关键的。

## 实时数据分析

相对于离线数据分析，实时数据分析也叫在线数据分析，针对的是对数据有实时要求的业务场景，如广告结算、订单结算等。目前，比较成熟的实时技术有 Storm 和 Spark Streaming。相比 Storm，Spark Streaming 本质上还是基于批量计算的。如果是对延迟很敏感的场景，应该使用 Storm。除了这两者，Flink 则是最近很火的一个分布式实时计算框架，其支持 Exactly Once 的语义，在大数据量下具有高吞吐、低延迟的优势，并且能够很好地支持状态管理和窗口统计，但其文档、API 管理平台等都还需要完善。

实时数据处理一般情况下都是基于增量处理的，相对于离线来说并不是可靠的，一旦出现故障（如集群崩溃）或者数据处理失败，则很难恢复数据或者修复异常数据。因此结合离线 + 实时是目前最普遍采用的数据处理方案。Lambda 架构就是一个结合离线和实时数据处理的架构方案。

此外，实时数据分析中还有一个很常见的场景：多维数据实时分析，即能够组合任意维度进行数据展示和分析。目前有两种解决此问题的方案：ROLAP 和 MOLAP。

- ROLAP：使用关系型数据库或者扩展的关系型数据库来管理数据仓库数据，以 Hive、Spark SQL、Presto 为代表。
- MOLAP：基于数据立方体的多维存储引擎，用空间换时间，把所有的分析情况都物化为物理表或者视图。以 Druid、Pinot 和 Kylin 为代表，不同于 ROLAP（Hive、Spark SQL），其原生地支持多维的数据查询。

如前面所述，ROLAP 的方案在大多数情况下支持用户离线数据分析，却满足不了实时需求，因此 MOLAP 是多维数据实时分析的常用方案。对于其中常用的 3 个框架，对比如表 1-2 所示。

表 1-2

	使用场景	语言	协议	特点
Druid	实时处理分析	Java	JSON	实时聚合
Pinot	实时处理分析	Java	JSON	实时聚合
Kylin	OLAP 分析引擎	Java	JDBC/OLAP	预处理、cache

其中，Druid 相对比较轻量，用的人较多，比较成熟。

## 数据即席分析

离线和实时数据分析产生的一些报表是给数据分析师、产品经理参考使用的，但是在很多情况下，线上的程序并不能满足这些需求方的需求。这时候就需要需求方自己对数据仓库进行查询统计。针对这些需求方，SQL 上手容易、易描述等特点决定了其可能是最合适的方式。因此提供一个 SQL 的即席查询工具能够大大提高数据分析师、产品经理的工作效率，如 Presto、Impala、Hive 等都是这种工具。如果想进一步提供给需求方更加直观的 UI 操作界面，可以搭建内部的 HUE，如图 1-5 所示。

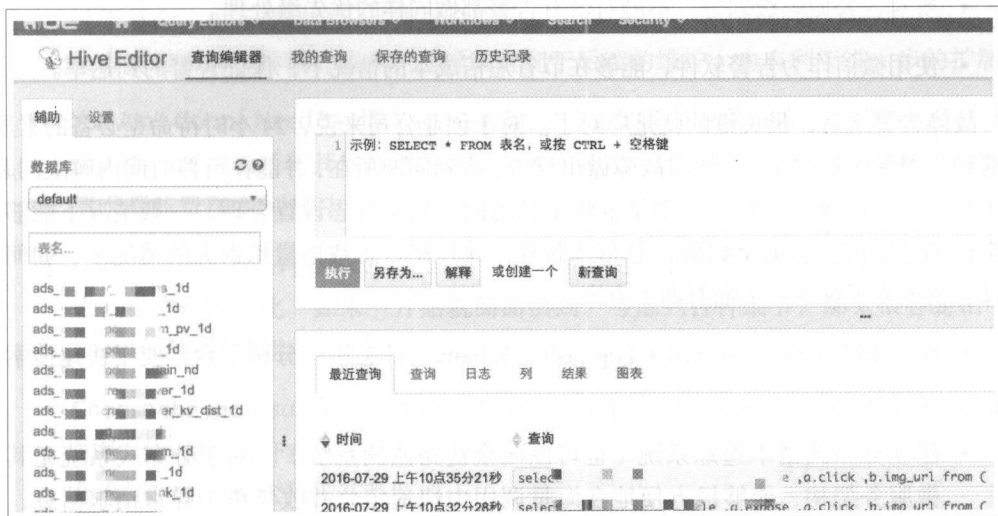


图 1-5

### 1.1.12 故障监控

对于面向用户的线上服务，发生故障是一件很严重的事情。因此，做好线上服务的故障检测告警是一件非常重要的事情。可以将故障监控分为以下两个层面的监控。

- 系统监控：**主要指对主机的带宽、CPU、内存、硬盘、I/O 等硬件资源的监控。可以使用 Nagios、Cacti 等开源软件进行监控。目前，市面上也有很多第三方服务能够提供对于主机资源的监控，如监控宝等。对于分布式服务集群（如 Hadoop、Storm、Kafka、Flume 等集群）的监控则可以使用 Ganglia。此外，小米开源的 OpenFalcon 也很不错，涵盖了系统监控、JVM 监控、应用监控等，也支持自定义的监控机制。
- 业务监控：**是在主机资源层面以上的监控，比如 APP 的 PV、UV 数据异常，交易失败等。需要业务中加入相关的监控代码，比如在异常抛出的地方加一段日志记录。

监控还有一个关键的步骤就是告警。告警的方式有很多种：邮件、IM、短信等。考虑到故障的重要性差异、告警的合理性、便于定位问题等因素，有以下建议。

- 告警日志要记录发生故障的机器 ID，尤其是在集群服务中，如果没有记录机器 ID，那么对于后续的问题定位会很困难。
- 要对告警做聚合，不要每一个故障都单独进行告警，这样会对工程师造成极大的困扰。
- 要对告警做等级划分，不能对所有告警都做同样的优先级处理。
- 使用微信作为告警软件，能够在节省短信成本的情况下，保证告警的到达率。

故障告警之后，最关键的就是应对了。对于创业公司来说，24 小时待命是必备的素质，当遇到告警的时候，需要尽快对故障做出反应，找到问题所在，并能在可控时间内解决问题。对于故障问题的排查，基本上都是依赖于日志的。只要日志设置合理，一般情况下能够很快定位到问题所在，但是如果是分布式服务，并且在日志数据量特别大的情况下，如何定位日志就成为了难题。下面有两个方案。

- 建立 ELK (Elasticsearch + Logstash + Kibana) 日志集中分析平台，便于快速搜索、定位日志。
- 建立分布式请求追踪系统（也可以叫全链路监测系统）。对于分布式系统尤其是微服务架构，能够极方便地在海量调用中快速定位并收集单个异常请求信息，也能快速定位一条请求链路的性能瓶颈。唯品会的 Mercury、阿里的鹰眼、新浪的 WatchMan、Twitter 开源的 Zipkin 基本都是基于 Google 的 Dapper 论文而来的。此外，Apache 正在孵化中的 HTrace 则是针对大的分布式系统，诸如 HDFS 文件系统、HBase 存储引擎而设计的分布式追踪方案。这里需要提到的一点是，如果你的微服务实现使用了 Spring Cloud，那么 Spring Cloud Sleuth 则是最佳的分布式追踪方案。

## 1.2 Java 后端技术概览

基于 1.1 节介绍的后端基础设施，可以从中引出其需要的技能列表。进一步细化可以得到更为具体的技能树，也是 Java 后端工程师技能树的一个总体概览图（见本书下载资源）。

本书知识点围绕此技能树进行展开。

## 1.2.1 软件开发的核心理念

此处所说的是软件开发应该遵循的一些核心理念。

- **Don't Repeat Yourself**: 这是软件开发的一个基础原则,即不要做重复性劳动,也是现在所说的“极客文化”的一种。代码重复、工作重复在软件开发中都是不合理的存在,利用各种手段消除这些重复是软件开发的一个核心工作准则。
- **Keep it simple stupid**: 即 KISS 原则。在做软件设计的工作中,很多时候都不要想得过于复杂,也不要过度设计和过早优化,用最简单且行之有效的方案也就避免了复杂方案带来的各种额外成本。这样既有利于后续的维护,也有利于进一步的扩展。
- **You Ain't Gonna Need It**: 即 YAGNI 原则。只需要将应用程序必需的功能包含进来,而不要试图添加任何其他你认为可能需要的功能。因为在一个软件中,往往 80% 的请求都花费在 20% 的功能上。
- **Done is better than perfect**: 在面对一个开发任务时,最佳的思路就是先把东西做出来,再去迭代优化。如果一开始就面面俱到,考虑到各种细节,那么很容易钻牛角尖而延误项目进度。
- **Choose the most suitable things**: 这是在做方案选择、技术选型时候的一个很重要的原则。在面对许多技术方案、开源实现的时候,务必做到不能盲目求新,要选择最合适的而非被吹得天花乱坠的。

## 1.2.2 软件开发的过程管理

在一个软件的生命周期中,除了开发还有其他步骤,也都需要掌握一些技术,分别介绍如下。

- **项目管理**: 项目管理对于一个软件的开发是非常重要的,能够保证项目进度有条不紊地进行,在可控的时间内以一定的质量交付。瀑布开发模型、螺旋开发模型是传统的项目管理模型。在互联网的开发工作中,敏捷开发则是比较受推崇的开发方式。所谓的敏捷开发即快速实现原型,然后快速迭代。Scrum 是目前普遍流行的敏捷开发方式之一。
- **测试驱动开发**: 在平时的开发过程中,目前比较流行也是行之有效的一种方式就是 Test Driven Develop,即测试驱动开发。这种方式的核心就是编写单元测试。简单来讲,就是先完成某一个功能的单元测试用例,然后在逐步消除测试用例的编译错误的过程中完成功能的开发。

- **持续集成**：某一个软件功能完成开发之后，后续还有测试、预发布、部署等过程。整个过程被称为集成，而持续集成指的是无须人工干预即可不断地进行这个过程。Jenkins、Quick Build 都是比较典型的持续集成工具。

### 1.2.3 日常开发常用工具

日常开发指的是一些日常需要掌握的技能、工具等，分别介绍如下。

- **编辑器**：开发中现在用得比较多的编辑器包括 Emacs、Vim 和 SublimeText。笔者用得最多的就是 SublimeText，基本能够满足自己的开发需求，包括编写脚本代码、查看代码文件等。Vim 和 Emacs 这两款编辑器相对 SublimeText 来说需要记住很多命令，有一定的上手门槛。
- **源码版本管理**：代码的版本管理工具由 CVS 到 SVN 再到现在的 Git，已经在事实上形成了以分布式版本管理为主的版本管理方案。基于 Git，可以采用 GitFlow 作为源码管理模型。
- **项目工具**：GitHub 是一个第三方 Git 中央仓库，是目前世界上最大的开源代码库，也能够作为私人的代码管理软件；Facebook 开源的 Phabricator 提供了非常强大的任务管理、Bug 管理、测试、代码管理等，但其上手门槛相对较高；禅道是国人开发的一款项目管理工具，但是其免费版功能有限；以 Tower.im 为代表的第三方项目管理服务也是一个可以选择的方案，风险在于数据都不再是私有的。

### 1.2.4 应用的运行环境

后端应用开发完成之后是需要部署到服务器上对外提供服务的。从最开始的直接在物理机上部署服务，到后来的虚拟环境、云环境，再到现在火热的容器，直至最近兴起的无服务器技术，都是为了让服务的运行环境能够更加便于建立、更容易维护、更容易扩展。

- **Linux**：说到后端服务器肯定绕不开 Linux。至少现在互联网的后端服务绝大多数都是部署在 Linux 的各种服务器版本中的。其中 CentOS、Ubuntu 以及 Debian 是用得比较多的版本。对于 Linux，需要熟练掌握的就是很多常用的 Shell 命令，如 ps、netstat、lsof、ss、df、dh 等。此外，很多性能分析命令，如 top、vmstat、iostat、sar 等，也需要熟练使用。
- **应用服务器**：就 Java 来讲，很多时候开发的都是 Web 应用，以 HTTP 协议对外提供服务。除了对性能要求比较苛刻的情况下需要自己构建 HTTP 服务外，大部分情况需要依赖于支持 Java 程序的应用服务器。目前最常用的有：Tomcat、Jetty。严格

来讲,这两者只是 Servlet 容器,真正的 Java EE 应用服务器如 JBoss、WebLogic 在互联网领域很少使用。当然,这些软件并没有提供 URL 重写、请求委托等 Web 服务器功能,还不足以担当完整 Web 服务器的角色。Nginx 则是目前最流行的 Web 服务器。

- **负载均衡**:在高并发流量环境下,后端服务会以集群的模式对外提供服务。在集群的前面,需要负载均衡器将请求分配到集群的各个节点上。LVS 是最流行的四层负载均衡软件,HAProxy 是另一个既支持四层又支持七层负载均衡的软件,Nginx 则是七层负载均衡最流行的解决方案。当然,性能最好的负载均衡方案是以 F5 为代表的硬件负载均衡,但由于成本昂贵,在互联网团队中很少使用。此外,这里需要补充的是,为了保证同等角色的服务的高可用,如 LVS 经常作为流量的入口,会部署多个 LVS 节点互为主备,防止一个节点挂掉导致服务不可用。而实现互为主备的技术目前用得最多的就是 Keepalived。
- **虚拟化**:虚拟化技术是前几年经常用来做私有云的一种技术,即将自己的物理主机通过虚拟化技术分裂为多个虚拟主机,以隔离资源。其中,VPS(虚拟专用服务器)的代表技术有微软的 Virtual Server、VMware 的 ESX Server、SWsoft 的 Virtuozzo。此外,OpenStack 提供的构建私有 IaaS 的功能、Cloud Foundry 提供的构建私有平台运行环境以及 Docker 带来的容器服务,都是虚拟化技术的一种。

### 1.2.5 常用第三方服务

虽然从根本上讲所有的软件服务都是可以自己开发或者部署到自己服务器上的,但是受限于成本、周期或者其他客观因素,很多服务需要第三方提供。

- **IaaS: Infrastructure as a Service**,是云计算最开始的一种模式,现在基本上所有的云服务商都提供 IaaS 服务。其中,全球最强大的云服务商是亚马逊的 AWS,国内的则当属阿里云。就目前来看,即使是强如 AWS 也会出现一些运维故障,国内的这些云服务商的服务健壮性、运维响应更是经常被人吐槽。就笔者自己的经历来看,2010 年左右,盛大云的云服务其实做得还不错,但后来由于种种原因市场份额越来越小。国内除了阿里云,UCloud 也是一家专注做云计算的比较靠谱的公司。此外,还有一个青云,定位略显高端,也是不错的选择。当然,现在这些云服务商早就不仅仅是 IaaS 了,也提供了很多 PaaS 服务。
- **PaaS: Platform as a Service**,即只需要提交代码到指定的运行环境,其他的诸如代码打包、部署、IP 绑定等都由平台完成。除了可以使用 Cloud Foundry 构建自己

的 PaaS 平台以外，现在最流行的第三方 PaaS 服务有新浪的 SAE、百度的 BAE 以及 Google 的 GAE。

- **域名**：有一个可以提供服务的应用后，域名也是一个必需的基础设施。一个好的域名不仅代表了企业的形象，也能够更加方便用户的记忆与传播。目前购买域名可以通过国外的 name.com、godaddy 以及国内的万网等。有了域名之后下一步就得进行备案，域名提供商一般都提供了配套服务，或者找代理也可以办下来。此外，对于域名的解析，域名提供商一般会内置解析功能，也可以使用独立的 DNS 服务，如 DNSPod。
- **CDN**：内容分发网络，即就近请求的一种技术实现。服务提供方将会被大量访问的内容在全国的多个节点都做缓存，这样当用户访问时就能够就近选择，从而减少网络传输延时，提高访问速度。国内目前七牛和又拍云都提供了不错的 CDN 服务，当然像阿里云、UCloud 这种综合云服务商也都提供了 CDN 服务。
- **邮件发送**：这主要依赖邮件服务器，通过 SMTP 协议就可以实现发送。可以选择自己搭建邮件服务器，也可以选择使用腾讯邮箱、网易邮箱等。
- **短信发送**：使用短信发送验证码、营销短信是很常见的应用场景。由于短信是需要运营商支持的，所以基本上都需要依赖第三方代理。市面上有很多短信网关代理。
- **消息推送**：在移动应用上，推送已经成为一个标配功能。目前个推应该是第三方推送服务中的佼佼者，而且由于其客户很多，在联盟唤醒（一系列 APP 形成一个联盟，属于此联盟的 APP 中只要有一个被用户启动，就会静默唤起其他成员 APP 的后台进程。）上有很大的优势。
- **开放平台**：通过开放平台，可以使用 OAuth 等协议获取用户在第三方平台上的信息，以实现第三方平台登录等。目前，微博、微信、QQ 是最常见的第三方登录方式，基本上都是使用 OAuth 协议为第三方开发者提供服务的。
- **支付接口**：支付接口是很多内置购买功能软件的必备组件。目前，接入最多的无非是支付宝和微信，二者都提供了开放平台供商家接入。当然，也有直接绑定银行卡支付的，此时需要走的就是银行或者银联的网关接口。

## 1.2.6 计算机基础知识

像数据结构、算法、计算机网络、操作系统、计算机组成原理这些计算机科学基础知识，不管对于后端还是其他领域，都是必备的技能，也是所有软件开发的基础。扎实的计



计算机科学基础才能让你在学习、使用某种技术开发软件、调试软件、排查问题时心里有底、有据可循。

- **数据结构**：数据结构是组成程序的基础。经典的数据结构有字符串、数组、链表、哈希表、树（二叉树、平衡树、红黑树、B 树）、堆栈、队列、图。
- **算法**：经典的排序和查找算法在平时的开发工作中经常会用到，如冒泡排序、插入排序、选择排序、归并排序、快速排序、希尔排序、堆排序以及二分查找等。此外，在函数 / 方法的算法实现中要注意递归和迭代各自的优缺点。而衡量算法性能的主要因素包含空间复杂度和时间复杂度。
- **业务相关算法**：除了上面的基本算法外，业务中还会经常涉及一些更复杂的算法，如压缩算法、LRU 缓存算法、缓存一致性、编译原理中的状态机等。此外，目前越来越火的机器学习中也有很多算法，在很多业务场景中有很大的用途，如用于文本分词的结巴分词和中科院 ICTCLAS，用于关键词提取的 TF-IDF 和 TextRank，用于计算文本相似度的主题模型、Word2Vec、余弦相似度以及欧几里得距离，用于文本分类的朴素贝叶斯，用于推荐的聚类、协同过滤、用户画像、隐语义模型等。
- **计算机网络**：TCP/IP 协议是网络最根本的协议，其七层 / 四层协议栈的设计都是非常精华的东西，连接的建立、断开以及连接的各种状态的转换都是排查、解决网络问题的根本依据。从 TCP/IP 往上，HTTP 协议是现在绝大多数后端应用对外提供的协议，发展到现在已经将要步入 HTTP 2.0 时代，带来了持久连接、连接复用等令人振奋的新特性。此外，基于 HTTP 的 HTTPS 协议由于安全性高，正在逐渐成为后端服务对外开放的主流协议。业务层面，基于 HTTP 协议的 RESTful 规范正成为对外接口的主流规范，而 OAuth 2.0 协议也成为开放平台对外的主流协议。除了 HTTP 之外，SMTP 是另一个基于 TCP/IP 的应用协议，主要用在发送邮件上。
- **设计模式**：在软件开发中，前人的经验形成了很多经典设计模式供我们使用，能够使软件实现可重用、可扩展、可维护。经典的工厂模式、简单工厂模式、单例模式、观察者模式、代理模式、建筑者模式、门面模式、适配器模式、装饰器模式在日常的很多开发场景下都具有很重要的意义。

### 1.2.7 数据处理相关技能

现在互联网的所有业务其实都是围绕数据来进行的，因而数据传输、数据存储、数据分析处理都成为关键环节。



- **高速缓存**: 目前用得最广泛的缓存软件 Redis 能够支持丰富的数据结构, 如字符串、列表、有序集合等多种数据的存储。了解缓存实现的原理、内存淘汰的策略能够更好地使用缓存。此外由于缓存的成本较高, 在使用缓存的时候一定要做好量化和存储优化工作。
- **数据库**: 掌握数据库的很大一个关键点就在于对索引的使用, 可以说, 正确地使用索引就基本等于掌握了数据库的使用。目前绝大多数数据库都使用 B 树作为索引的数据结构, 目的就是为利用磁盘顺序读 / 写的特性。不同的数据库由于本身设计目的的不同, 都有自己独特的优势, 如 MongoDB 天然支持 Sharding, 但受限 NoSQL, 在重事务、有关联关系的场景下并不适用, 而 HBase 使用 LSM 作为底层数据结构, 牺牲了读性能来换取高速的写性能。
- **搜索引擎**: 搜索引擎主要应对全文检索以及多维度查询的业务场景。掌握搜索引擎使用的数据结构、集群方式、配置的关键点, 有助于更好地使用搜索引擎服务于业务应用。
- **消息队列**: 消息队列有两种角色, 即生产者和消费者, 两种角色对于消息队列的需求也不一样。其中, 对于消费者来说, 消息消费的方式包括发布-订阅和队列两种。消息队列在语义保证上分为 At Most Once、At Least Once、Exactly Once 共 3 种模式, 需要根据特定的业务场景选择合适的语义保证。此外, 消息队列对于高可用、消息安全的保证决定了此消息队列的可靠性。
- **数据存储和处理**: 数据存储下来最终还是要用于做分析和处理的。数据的处理分为离线处理和实时处理。离线处理的优势在于能够处理大量数据, 但是一般会有  $T+1$  的延迟, 适用于计算量大但对于结果允许有延时的场景。对于离线数据分析, 还有一个很关键的问题就是数据倾斜。数据倾斜指的是 Region 数据分布不均, 造成有些节点负载很低, 而有些却负载很高, 从而影响整体的性能。因此, 处理好数据倾斜问题对于离线数据处理是很关键的。而实时处理一般是流式处理方式, 适用于数据能够转换为数据流, 对于结果要求具有及时性的场景。对于实时数据分析, 需要注意的就是实时数据处理结果写入存储的时候要考虑并发的问題, 虽然对于 Storm 的 Bolt 程序来说不会有并发问题, 但是写入的存储介质是会面临多任务同时读 / 写的。通常的方案就是采用时间窗口的方式对数据做缓冲后批量写入。
- **数据同步**: 数据仓库的数据来源除了直接的日志外, 还有一个关键点就是业务数据库。数据从业务数据库到数据仓库的过程被称为数据同步。有基于 SQL 的同步方案, 也有基于 MySQL Binlog 的增量同步方案。

## 1.2.8 Java 编程知识

对于 Java 方面的技能来说，主要分两大部分，包括 Java 编程和 JVM。先来看一下 Java 编程部分，这也是 Java 工程师最基础的技能。

- **IDE**：目前用得最多的 Java IDE 当属 Eclipse 和 IntelliJ IDEA。前者是老牌 IDE，逐步淘汰了 Jbuilder 以及 Netbeans，占领了大部分 Java IDE 市场。后者则是后起之秀，由于其增量编译、智能分析代码等带来的性能提升，现在已经得到了大规模使用，大有取代 Eclipse 之势。
- **核心语法**：目前用得最多的当属 JDK 6 的 Java 语法。而 Java 7 则又引入了 try-with-resource、switch string、diamonds 等语法；Java 8 则又引入了 Lambda、Stream 等语法。
- **集合类**：集合类是 Java 语言中非常精华的部分，包括 HashMap、ArrayList、LinkedList、HashSet、TreeSet 以及线程安全的 ConcurrentHashMap、ConcurrentLinkedQueue 等线程安全集合。了解它们的实现原理、查询、修改的性能和使用场景是非常必要的。
- **工具类**：Google Guava、Apache Commons、FastJson 提供了很多 JDK 本身没有的工具类、集合等。此外，ASM 字节码操作以及 CGLIB 代码生成能够提供更底层的 Java 编程功能。
- **高级特性**：抛开 Java 核心的基本编程，并发编程、泛型、网络编程、序列化 RPC 都属于 Java 的高级编程特性。其中并发编程需要掌握 Executors 提供的各种并发工具、Java 7 带来的 Fork/Join 框架以及 CountdownLatch、Semaphore、CyclicBarrier 等同步工具；网络编程要区分好 BIO、NIO 以及 AIO；序列化中除了 JDK 自带的序列化实现外，Protobuf 和 Kryo 是比较高效的第三方实现；RPC 的实现中，Thrift、Hessian、Dubbo 以及 RMI 则是比较常用的几个协议，其中的 Hessian 是基于 HTTP 协议的，Dubbo 是基于 TCP 协议的，而 Thrift 则同时支持两种协议。
- **Java EE**：Java EE 现在是 Java 应用最普遍的一个领域。Servlet 是 Java EE 中最根本的组件之一。而 Servlet 3.0 带来的异步 Servlet 提高了其处理请求的性能。
- **项目构建**：目前用得最多的 Java 项目构建工具包括 Maven 和 Gradle，它们提供了源码包依赖管理、编译、打包、部署等一系列功能。
- **编程框架**：Spring 是 Java 编程中避不开的一个框架，发展到现在除了 Spring 核心的 IoC、AOP 之外，Spring MVC、Spring Data、Spring Cloud 等都给 Java 开发者们带来了开发上的便利，大大提高了开发效率。除此之外，ORM 框架 MyBatis 也是 Java 领域比较火的框架之一，实现了数据库记录到 Java 对象的映射操作。此外，Jersey 提供了从客户端到服务器端的一整套符合 RESTful 规范的开发框架。

- **测试**：测试是任何编程都需要的一步。黑盒测试主要指的是通常进行的功能测试，白盒测试则主要指的是对代码功能、质量进行的测试。此外，关键的单元测试则是开发工程师需要着重注意的地方，“测试驱动开发”的理念也是值得推崇的开发方式。JUnit 是目前 Java 中实现单元测试的主流方案。

一般来说掌握了上面所述的 Java 编程技能就能够应付大多数编程工作，但是如果在代码层面已经做到最大努力却还是达不到性能要求的时候，就需要在 JVM 虚拟机层面做一些努力。可以说掌握 JVM 相关技术是 Java 开发进阶的一个关键步骤。

- **虚拟机实现**：Java 的虚拟机实现除了我们常用的 HotSpot 外，还有 JRockit、J9 以及移动平台的 Dalvik。我们通常所描述的 JVM 优化绝大多数是针对 HotSpot 虚拟机来说的。
- **类加载机制**：JVM 的类加载机制遵循双亲委派原则，即当前类加载器需要先去请求父加载器加载当前类，无法完成才自己去尝试进行加载。OSGI 框架则打破了此机制，采用了平等的、网状的类加载机制，以实现模块化的加载方案。
- **运行时内存组成**：程序计数器、堆栈、方法区、堆、堆外内存，共同组成了 JVM 的运行时内存。
- **Java 内存模型**：Java 的主内存 + 线程私有内存的模型是线程安全问题产生的根本。
- **GC 原理和调优**：与 C、C++ 这些语言相比，GC 是 Java 的优势，但因为 GC 的细节被 JVM 屏蔽了，故在对内存、性能要求非常苛刻的情况下难以进行自由控制，从某种程度上说这也是劣势。如果想在某些场景下发挥 GC 的最大性能，能做的就是对 GC 的各种参数做优化配置，如新生代和老年代的垃圾回收器选择、各种垃圾回收参数的配置等。此外，很多时候由于代码质量或者外部客观因素，造成了 JVM 频繁 GC，需要使用相关工具快速进行问题定位和解决。
- **性能调优和监控工具**：JDK 自带了很多强大的调优和监控工具，包括 jmap、jstack、jcmd、JConsole、jinfo 等。此外，BTrace 是一款非常强大的在线问题动态排查工具，能够无须重启 Java 进程即可动态地插入一些代码逻辑，从而拦截代码执行逻辑打印日志并排查问题。

### 1.2.9 系统架构演化

一个应用从零开始一般会经历单体应用、垂直应用到分布式服务架构的演化，如图 1-6 所示。

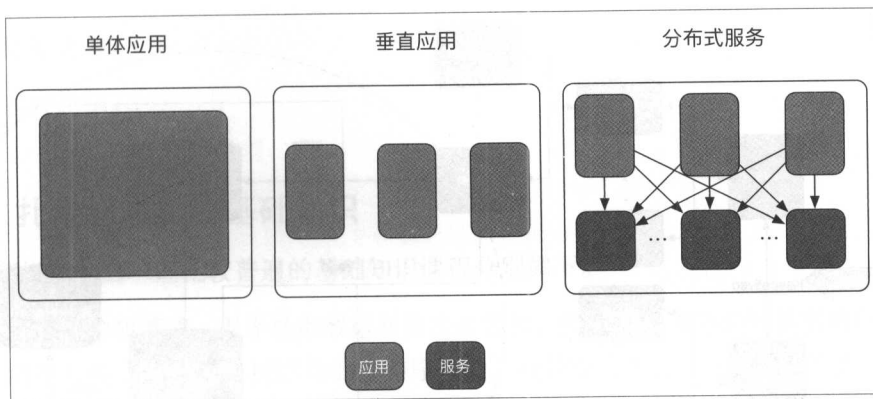


图 1-6

- **单体应用**：当应用规模、团队规模比较小的时候，只需要一个包括了所有功能的应用。这样可减少部署节点，也减少部署成本。此时，对数据库的 ORM 操作是架构实现的关键点。
- **垂直应用**：当应用的用户规模越来越大，请求量越来越高的时候，单体应用增加节点带来的资源浪费会凸现出来，因为绝大多数接口请求量并不是特别大，根本没必要扩充到多个节点，完全可以将单体应用拆分成互不相关的几个应用，分别对外提供服务。此时，加速每个应用开发的 MVC 框架是架构实现的关键点。
- **分布式服务**：当垂直应用越来越多时，应用之间的交互不可避免。要考虑抽离核心业务单独部署，逐渐形成稳定的服务中心。而随着团队规模的相应扩大，服务会随着团队的增多变得越来越多，粒度会变得越来越小，也就逐步形成了分布式服务的架构，而当粒度细到某种程度、服务数量多到一定程度，则可以称之为微服务。即在设计好业务边界之后将原来的单体应用分解成一个个细粒度的服务，彼此之间通过某种方式进行通信。微服务架构的关键在于如何做好服务的治理、调度、维护工作。目前，Dubbo 算是微服务架构中用得比较多的框架，但 Dubbo 仅仅解决了微服务架构中的一部分问题。另外，Spring Cloud 则基本上涵盖了微服务架构的各个方面。

### 1.2.10 典型的部署架构

对于 Web 应用来说，LVS+Nginx+Tomcat+MySQL+Redis 即可构成一个简单、通用的部署架构，如图 1-7 所示。

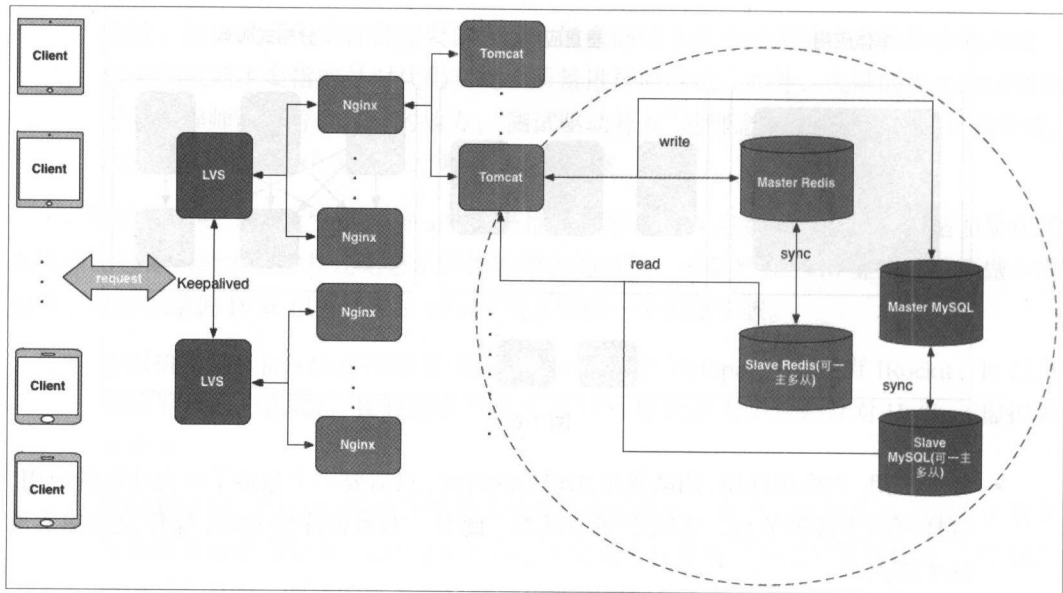


图 1-7

- LVS 作为最前置的节点，负责在网络第四层转发流量、负载均衡。
- 多个 LVS 使用 Keepalived 互为主备实现高可用。
- Nginx 作为反向代理，负责在网络第七层转发流量、负载均衡。
- Tomcat 作为业务容器，主要的业务代码都放在这里面。
- Redis 作为缓存，隔离高并发请求和后端数据库。
- MySQL 以主从模式对数据做持久化。

其中，虚线部分是数据库层，采用的是主从模式。也可以使用 Redis Cluster（Codis 等）以及 MySQL Cluster（Cobar 等）来替换。

### 1.3 如何学习后端技术

学习后端技术和学习其他技术并没有什么大的不同。因此，本章题目换作“如何学习技术”也是讲得通的。概括来讲，有以下几点建议。

- 扎实的计算机基础知识。
- 知其然更要知其所以然。
- 动手实践。

- 频繁练习。
- 自我总结。

### 1.3.1 扎实的计算机基础知识

计算机专业的一些比较普遍的基础知识课程分别如下。

- **数据结构和算法**：程序是由数据和算法组成的，因此这两部分是计算机软件的基础，诸如 B 树、哈希表、栈以及七大排序算法、查找算法等，在很多软件代码中都可以看到。有时候，一名优秀工程师和一名普通工程师的区别也就在于是否能够使用合适的数据结构和算法。
- **计算机操作系统**：操作系统可以说是一个集大成于一身的软件程序。资源调度、任务调度、I/O 调度、进程通信等，每一个设计都是精华，也是很多其他应用软件设计的思想来源。
- **计算机网络**：目前绝大多数有数据传输功能的程序都离不开网络。网络七层或者四层协议栈的设计非常精妙。了解网络连接如何建立、断开以及每个连接状态的意义，都有助于对程序网络问题的排查。
- **计算机组成原理**：这是计算机底层的设计，也是计算机运行的基础。了解这些有助于消除计算机的神秘感，毕竟符合冯·诺依曼原理的计算机无非就是存储数据、程序按序运行。

这里不得不说的有一点是，虽然现在各大高校的计算机专业课程比较落后，但是这些基础课程是计算机专业的基础，因此不管怎样都应该扎实掌握。也许上学的时候你感觉不到有多大的用处，但是进入实际的开发工作中，是否能够掌握扎实的基础知识往往决定了一名开发工程师的上限，这也是很多大的互联网公司无论是校招还是社招，都侧重于一些底层知识考察的原因。毕竟，你会使用什么东西只能决定你的下限，而你的基础知识和学习能力才决定了你的上限。

此外，其实很多平时开发中用到的技术都可以关联到这些基础知识。比如，我们经常为了提高查询性能而使用的缓存技术，以及为了兼容 CPU 和 I/O 速度不匹配而设计的 CPU Cache 就是同样的东西；操作系统中的进程间通信方式和服务之间的异步 / 同步通信也是差不多的道理。诸如此类，其实计算机科学基础凝聚了很多精华的设计，无论是计算机硬件架构、计算机操作系统还是计算网络。

### 1.3.2 知其然更要知其所以然

经常遇到来面试的工程师，在简历上写了很多项目，也用很多技术，怎么看都是非常不错的候选人，但是面试一旦深入到原理或者优化层面，很多工程师甚至是一些公司的资深架构师都会支支吾吾，答非所问或者说是没关注过。听到最多的解释就是业务压力太大，没有时间去研究。其实，从笔者自己的经历来看，业务忙是原因，但是没有时间却肯定是借口，毕竟阅读一个项目的源码虽然比较耗费时间，但是去网上看看已有的原理分析其实是花不了太多时间的。归根结底，还是没有一种知其然更要知其所以然的基本意识。很多东西，你学会了使用会很兴奋，但是你有没有想过这么令人兴奋的功能是如何实现的呢？最简单的例子，Java 中的 HashMap，大家都在用，但是它到底是怎么实现的呢？很多人甚至都不知道它和数据结构课程上的哈希表是什么关系，更别说知道解决冲突的方法了。

这也可以映射到现在的一种现象：很多开发工程师工作了很久，看似经验丰富，但基本都是拿着一年的工作经验重复  $n$  年。基本上每一份工作，每一个项目都在做重复劳动，而且不去考虑如何避免重复劳动。

这也涉及技术的广度和深度的问题。就笔者来看，对于刚刚毕业或者刚刚入职的工程师来说，首要的是深度问题，只有你在某一领域有了深入的研究和造诣，你才能融会贯通，迅速地扩大自己的知识面，在广度上做到突破。而对于有一定工作经验的工程师来说，虽然深度不是那么必需，但是遇到的问题、新学到的东西还是刨根问底才好，否则一旦出现问题再去亡羊补牢会非常被动，也不利于自己的技术发展。毕竟，一个什么都做过、什么都用过却什么也不精通的人很容易被替代。

### 1.3.3 动手实践

一种学习技术的最佳实践就是“项目驱动型学习”，也就是动手实践。很多技术，只看书，你会觉得云里雾里，看了就忘，必须要经过自己的实践或者项目中使用到了这种技术，你才能很快地掌握并熟练。此外，现在朋友圈、微博上充斥着各种所谓的干货，很多人阅读大量的资料自以为收获满满，其实对于里面的知识根本就没有实践过，甚至有时候就是感叹一下别人真厉害而已。真正的干货是需要自己消化的，消化最好的模式就是实践，无论是对资料中的例子，还是一笔带过的知识，都是如此。

### 1.3.4 频繁练习

动手实践能够让你快速入门，但只有频繁练习才能让你熟练使用。



“一万小时”理论讲的是任何一个行业都至少需要一万个小时的实践才能成为专家。先不去讨论此理论是否正确，可以想想当你长时间不写代码或者不用某个技术后再去做相关的开发，那种生涩、陌生感想必是无疑的。可见对于研发这个角色，频繁练习是多么重要。

当然，这里的频繁练习并非指重复劳动，而应该是带着自己的思考去练习，多想一下为什么这么做，有没有更好的方式。

### 1.3.5 持续学习

“活到老，学到老”这句话用在程序员这个职业上再合适不过了。IT 技术尤其是互联网开发中的技术，迭代速度是非常快的。也许你今天学的知识，过不了几年就被抛弃。虽然相比前端技术，后端技术算是比较稳定的，但与其他行业相比，迭代速度还是非常快的，像 Struts 这种当年火得一塌糊涂的技术现在也过时了。因此，一定要对新事物、新技术具有敏感性，要不断地涉猎业界最新的知识点，扩充自己的知识库。

这里还需要提到的一点就是，要“逃离舒适区”。人们对自己熟悉的东西会感到亲切，对自己熟练掌握的技术一般也能够自信地使用，然而当需要使用自己没接触过的技术时，很多人就望而却步、不敢尝试了，进而也就丧失了学习新知识、扩充自己知识库的机会。最好的方式应该是敢于“逃离舒适区”、敢于使用新技术，这样才能让自己具有持续的学习兴趣，促进自己的持续进步。

### 1.3.6 自我总结

相信很多人在平常的工作中，经常会遇到一些问题，然后通过查阅网上资料、询问同事、翻看源码等手段解决了，当再次遇到类似的问题甚至相同的问题时，还是一头雾水。先不提记忆力的问题，造成这种情形的很大一个原因就是没有总结。当然，这里的总结不仅仅指的是把你平时遇到的问题记录下来，更深一层的是要找到问题发生的本质原因，如何避免发生同样的问题，从中有什么启发和收获等。再进一步，则需要经常将自己一段时间内的知识收获整理成体系或者融入自己的知识体系中，这样才能举一反三，遇到相同的问题可以有据可循。

而自我总结的方式包括记笔记、写博客、做分享。其中，相比记笔记来说，写博客、做分享是笔者更为推荐的方式。毕竟，和别人交流一方面能促使你对总结质量的把控，另一方面分享知识给别人带来的“荣誉感”反过来会产生某种正向反馈让你更加乐于总结和分享。



### 1.3.7 如何学习一门新技术

上面主要讲述了宏观层面的如何学习技术，而具体到学习某一新技术，其实也是有法可循的，如图 1-8 所示。

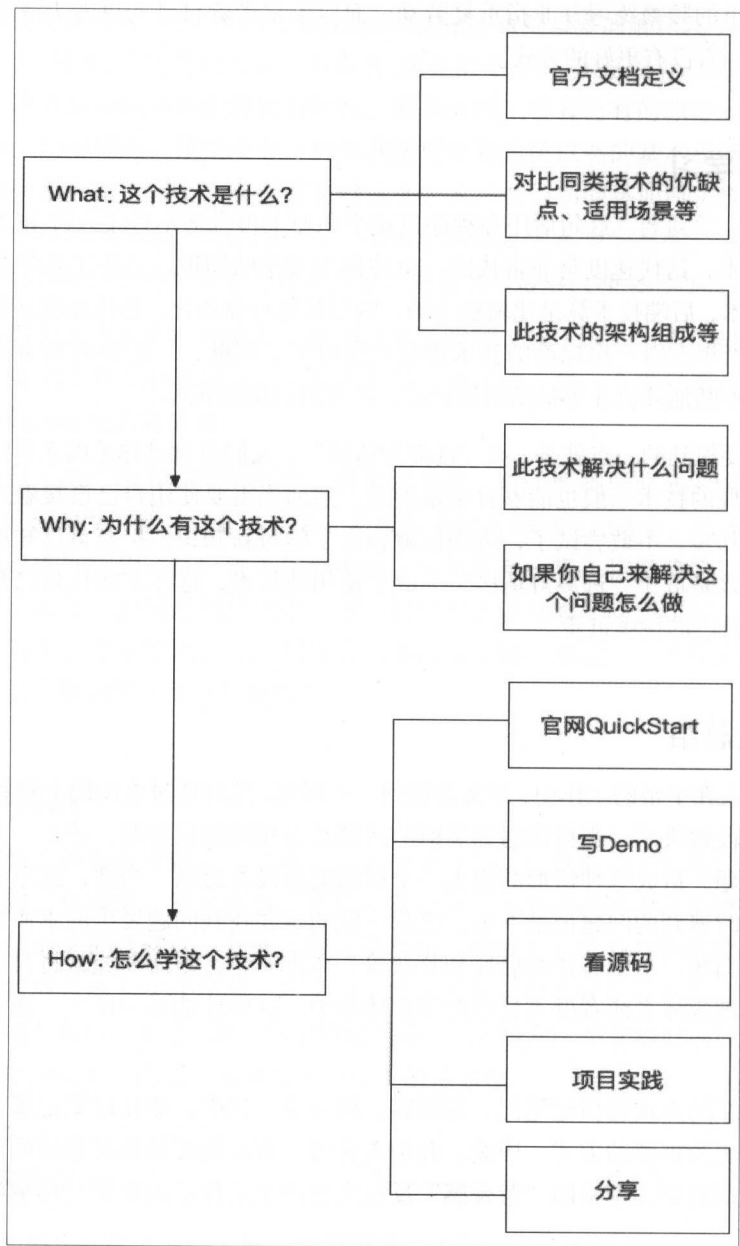


图 1-8

由于很多技术的模块非常多，源码也非常复杂，很多时候看源码会陷进去越看越不得章法，因此这里着重说明一下“看源码”的典型流程。

- 1) 阅读该技术的架构文档，了解其总体架构和组成。
- 2) 根据总体架构，将源码文件以模块或者上下层级进行分类。
- 3) 从未阅读过的模块中选择最独立（依赖性最小）的模块代码读起。
- 4) 阅读此模块的功能介绍文档。
- 5) 阅读此模块的源代码。
- 6) 一边阅读一边整理调用关系（以表或者树的形式）。
- 7) 转到第3)步。

### 1.3.8 小结

程序员是一个金字塔结构的职业体系，越往上，人越少也越难达到，如图1-9所示。

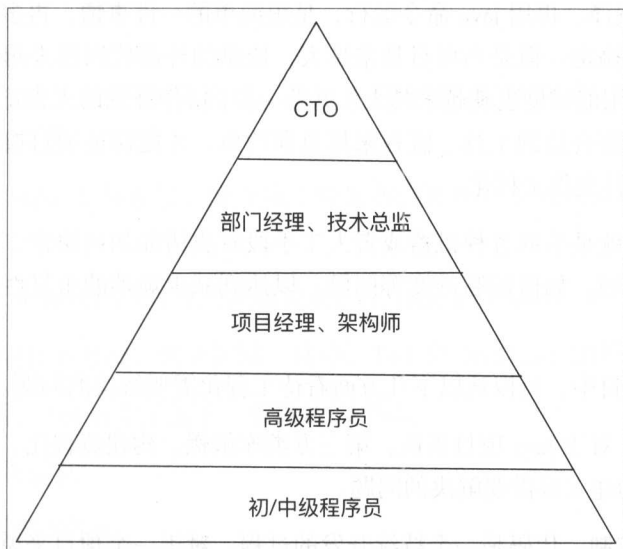


图 1-9

走到金字塔顶部需要不断的学习和进步，包括正确的态度、正确的方法以及持续的努力。本章所述只是笔者自己的体会，也是自己一直在践行的东西。除此之外，肯定还有很多其他优秀的方法和思想能够促进这个过程。

## 第 2 章

# Java 项目与工程化

可能大家听过一种说法，就是厉害的程序员是直接利用记事本软件来编写程序的。暂且不去对这句话辨伪，但可以肯定的一点是，单单使用记事本软件来编写程序不可能搞定所有的应用开发，尤其是大型应用的开发。试想，一个简单的小程序，用一个记事本软件写完，然后用 `javac` 命令编译，再用 `java` 命令运行，是很简单的一件事情，再多加点功能，厉害点的程序员也能轻松搞定。但是当项目越来越大，依赖的外部代码越来越多，单单用记事本软件来开发这个应用的难度也就越来越大。更进一步，协作开发的人也正在变得越来越多，人员之间的协作需要合适的工具、流程来规范和约束，才能保证项目按时保质地完成。这时候，就需要让项目变得工程化。

所谓的工程化就是采取各种机器或者人工手段解决诸如如何协作、如何规范化、如何保证质量、如何计划、如何控制进度等问题，以达到减少烦琐的重复性劳动，使得流程高效并且可重复。

具体到 Java 项目中，可以从以下几方面看待工程化需要解决的问题。

- **项目构建：**对于 Java 项目来说，第三方类库依赖、构建流程化、项目结构标准化等都是项目构建工具需要解决的问题。
- **代码版本控制：**代码是一个持续开发的过程，对于一个项目来说，在这个过程中还会存在新的需求、版本迭代、Bug 修复等很多小的分支。此外，项目多个成员的代码如何集成在一起、如何防止代码文件冲突，这些都是需要代码版本控制解决的问题。
- **代码质量保证：**这是开发工作中非常重要的一点，需要制定代码规范，保证代码的可读性、可维护性。同时需要用一定的手段消除代码的 Bug，提升应用的服务质量。

本章就主要针对这 3 方面做具体的阐述。

## 2.1 项目构建

对于一个完整的项目来说，一般都有很多的类、包，如果是 Web 工程则还有很多 JSP、资源文件。这时候如果只用 JDK 自带的工具编译和运行，是非常困难的一件事。再者，编译源代码只是软件开发过程的一个方面，更重要的是要把软件发布到生产环境中产生商业价值。所以，编译代码之后，还有测试、分析代码质量、部署等步骤要做。整个过程进行自动化操作是很有必要的。

这时候就需要一个 Java 的工程 / 项目构建工具。这里所谓的项目构建指的是完成工程发布流程需要的一系列步骤，包括编译、测试、打包、部署等。虽然用 Eclipse 和 IntelliJ 这些 IDE 能解决这个问题，但是受限于这些 IDE 体积庞大且基本上都是 GUI 的，而后端应用的运行环境几乎都没有显示器，所以很多时候还是需要一些专门做项目构建的工具来支持的。

### 2.1.1 传统构建工具——Ant

Ant 的全称是 Another Neat Tool，意为另一个好用的工具。

#### 介绍

Ant 是 Java 中比较常用的项目构建工具。其构建过程包括编译、测试和部署等，概括来看具有以下 3 个特点。

- 和传统的 Make 工具相似，能完成工程发布流程中一系列机械工作，并且具有良好的跨平台特性。
- 使用 XML 来表述构建过程与依赖关系，用 Task 替代 Shell，语义清晰，便于维护。
- 具有强大的任务系统，便于扩展。其中，Task 以 Java Class 的形式存在。

为了方便使用，Ant 自带了如下一些默认的 Task。

- echo：输出信息。
- mkdir：创建文件夹。
- exec：执行 Shell 命令。
- delete：删除文件。
- copy：复制文件。

通过组合这些默认 Task 和自己实现的 Task 就能够完成 Java 项目的构建任务。

## 使用示例

使用 Ant 需要编写 build.xml 来配置任务流程。当然，也可以通过 -f 参数指定其他配置文件作为任务流程描述文件。一个 Ant 的配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="helloWorld" basedir="." default="usage">
  <property name="mvn" value="mvn"/>
  <property name="script.lock" value="/build_home/scripts/lock"/>

  <target name="usage" description="Prints out instructions">
    <echo message=" 使用 'lock' 加锁 "/>
    <echo message=" 使用 'unlock' 解锁 "/>
    <antcall target="compile">
      <param name="profile" value="test"/>
    </antcall>
  </target>

  <target name="lock">
    <exec dir="${basedir}" executable="${script.lock}"
errorproperty="lock.err"/>
    <fail message="u can use 'ant unlco to force redeploy'...">
      <condition>
        <contains string="${lock.err}" substring="locked"/>
      </condition>
    </fail>
  </target>

  <target name="unlock">
    <delete file="${basedir}/.lock"/>
  </target>

  <target name="compile">
    <echo message=" 编译开始 "/>
    <exec dir="${basedir}" executable="${mvn}" failonerror="true">
      <arg line="compile -P ${profile}"/>
    </exec>
    <exec executable="${mvn}" failonerror="true">
      <arg value="war:exploded"/>
    </exec>
  </target>

  <!-- 逻辑判断 -->
  <target name="testIf" depends="check" if="flag">
    <echo message="if..." />
  </target>

  <target name="testUnless" depends="check" unless="flag">
    <echo message="unless..." />
  </target>
```

```

<target name="check">
  <condition property="flag">
    <or>
      <and>
        <isset property="name"/>
        <equals arg1="${version}" arg2="1.0" />
      </and>
      <available file="/project.version" type="file"/>
    </or>
  </condition>
</target>
<!-- 逻辑判断 end-->

```

```
</project>
```

可见, Ant 使用顶级元素 `<project>` 描述整个工程, 使用 `<property>` 描述全局属性, 使用 `<target>` 定义工程中的 `target` 以及 `target` 间的依赖, 在 `target` 中定义 Task 的执行流程。使用 `antcall` 来调用 `target`, 通过其子节点 `param` 传递参数。

此外, 在很多场景下需要用到逻辑判断, 如 `if` 等。Ant 中的 `if` 如上面的例子所示, 是需要搭配 `target` 和 `condition` 使用的。上面配置中最后的逻辑判断部分, 类似如下伪代码:

```

if (name != null && version.equals("1.0")) || fileExist("/project.
version") ){
    echo "if..."
}else{
    echo "unless..."
}

```

执行 `ant [target]` 即可执行任务流程。

## 提示

- 使用 Ant 时, 一个常见的需求就是通过命令行给 Ant 传递参数, 可以通过 `-Dname=value` 这种形式来传递, 在 `build.xml` 中通过 `${name}` 来引用即可。
- 对于配置文件中重复出现的元素, 可以通过 `refid` 引用, 减少重复配置。

```

<project>
  <path id="project.class.path">
    <pathelement location="lib/" />
    <pathelement location="${java.class.path}" />
  </path>

  <target ...>
    <rmic ...>
      <classpath refid="project.class.path" />
    </rmic>
  </target>

```

```

    <target ...>
      <javac ...>
        <classpath refid="project.class.path"/>
      </javac>
    </target>
  </project>

```

## 2.1.2 主流构建工具——Maven

Maven 是继 Ant 后出现的一款基于约定优于配置原则的项目构建工具。

### 介绍

上面所说的“约定优于配置”指的是约定的一些规范无须再配置，例如其约定好的生命周期、项目结构等。当然，Maven 也提供了打破默认约定的配置方法。

概括来讲，Maven 具有以下功能。

- **依赖管理：**Maven 能够帮助我们解决软件包依赖的管理问题，不再需要提交大量的 jar 包，引入第三方库也不需要关心其依赖。
- **规范目录结构：**标准的目录结构有助于项目构建的标准化，使得项目更整洁，还可以通过配置 profile 根据环境的不同读取不同的配置文件。
- **可以通过每次发布都更新版本号以及统一依赖配置文件来规范软件包的发布。**
- **完整的项目构建阶段：**Maven 能够对项目完整阶段进行构建。
- **支持多种插件：**面向不同类型的工程项目提供不同的插件。
- **方便集成：**能够集成在 IDE 中方便使用，和其他自动化构建工具也都能配合使用。

可见，相比 Ant，Maven 提供了更加强大和规范的功能。

### 配置文件

Maven 基于 POM (Project Object Model) 进行。一个项目所有的配置都被放置在 pom.xml 文件中，包括定义项目的类型、名字，管理依赖关系，定制插件的行为等。如下：

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">

```

```

<modelVersion>4.0.0</modelVersion>

<groupId>me.rowkey</groupId>
<artifactId>test</artifactId>
<version>1.0.0</version>
<packaging>jar</packaging>

<name>antares</name>
<url>http://maven.apache.org</url>

<repositories>
  <repository>
    <id>nexus-suishen</id>
    <name>Nexus suishen</name>
    <url>http://maven.etch.cn/nexus/content/groups/public/</url>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>always</updatePolicy>
      <checksumPolicy>warn</checksumPolicy>
    </snapshots>
  </repository>
</repositories>

<properties>
  <slf4j.version>1.7.21</slf4j.version>
</properties>

<dependencies>

  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
  </dependency>

  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
  </dependency>

</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>

```



```

        <source>1.7</source>
        <target>1.7</target>
    </configuration>
</plugin>
</plugins>
</build>
</project>

```

其中,

- Maven 使用 groupId:artifactId:version 三者来标识一个唯一的二进制版本, 可以缩写为 GAV。
- packaging 代表打包方式, 可选的值有 pom、jar、war、ear、custom, 默认值是 jar。
- properties 是全局属性的配置。
- dependencies 是对于依赖的管理。
- plugins 是对于插件的管理。

此外, 可以通过 parent 实现 POM 的继承以完成统一配置管理, 子 POM 中的配置优先级高于父 POM。

```

<?xml version=" 1.0" encoding=" UTF-8" ?>

<project>
...

<parent>

    <artifactId>suishen-parent</artifactId>

    <groupId>suishen</groupId>

    <version>1.0</version>

</parent>
...

</project>

```

可以继承的元素如下。

- groupId, version。
- Project Config。

- Dependencies。
- Plugin configuration。

此外，<dependencyManagement> 和 <pluginManagement> 可以统一做依赖和插件的配置管理，不同于 <dependencies> 和 <plugins> 的是，如果子 POM 中没有声明 <dependency> 和 <plugin> 则并不生效。

## 标准 Web 项目结构

在 Maven 中，一个 Web 项目的标准结构，如图 2-1 所示。

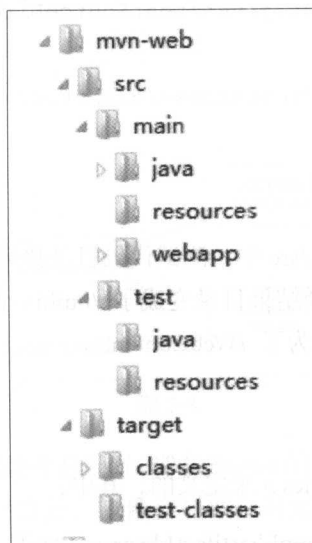


图 2-1

其中，

- src/main/java Java 代码目录
- src/main/resources 配置文件目录
- src/main/webapp webapp 根目录
- src/test/java 测试代码目录
- src/test/resources 测试配置目录
- target/classes 代码编译结果目标目录
- target/test-classes 测试代码编译结果目标目录

当然，结构是可以自定义的：

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <configuration>
        <warSourceDirectory>WebContent</warSourceDirectory>
      </configuration>
    </plugin>
  </plugins>
  <sourceDirectory>src</sourceDirectory>
  <testSourceDirectory>test/java</testSourceDirectory>
  <testResources>
    <testResource>
      <directory>test/resources</directory>
    </testResource>
  </testResources>
  <directory>build</directory>
</build>
```

这里，Java 代码目录移到了 `./src` 中，测试代码目录移到了 `./test/java` 中，测试资源也移到了 `./test/resources` 中，同时编译结果目录变成了 `./build`。此外，在 `maven-war-plugin` 中，也把 Web 目录的 war 源码目录改为了 `./WebContent`。

## 依赖管理

依赖管理是通过 `<dependencies>` 来定义的，其中：

- 一项 jar 包依赖可以由 `groupId:artifactId:version` 标识。
- 完整的标识为：`groupId:artifactId:type:classifier:version`。
- 依赖在编译部署中参与的情况可以由 `scope` 来指定，分为 `compile`、`test`、`provided`、`system`、`import`，默认值是 `compile`。其中的 `import` 是在 Maven 2.0.9 后引入的，仅仅支持在 `<dependencyManagement>` 中使用，导入外部的依赖版本管理。
- 依赖是一个树状结构，采用最近依赖原则，也可以通过 `exclusions` 标签来排除一些包。这里的最近依赖指的是在依赖树中，离当前节点最近的依赖优先级高，同样远时第一个优先。

依赖下载的过程一般如图 2-2 所示。

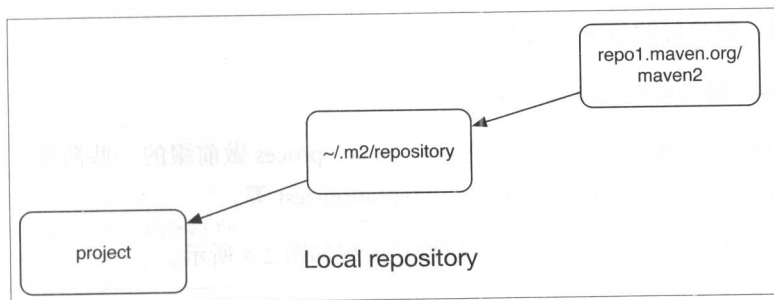


图 2-2

但上面的示例配置中，如果有一个 repository 的配置，那么依赖下载的过程会发生改变，如图 2-3 所示。

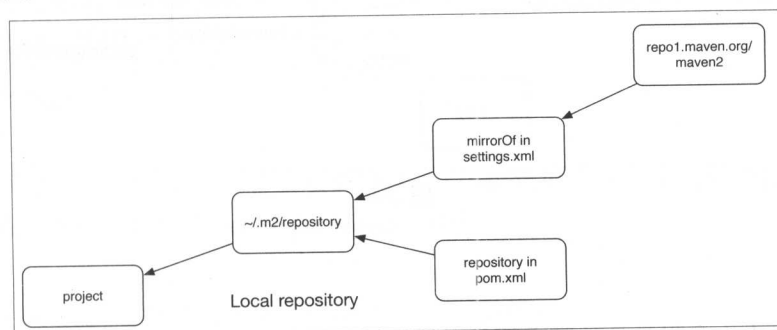


图 2-3

此外，Maven 中还有一个镜像库的配置，即在 Maven 的 settings.xml 中配置 Maven 镜像库。和 pom.xml 中的 repository 不同的是，镜像会拦截住对远程中央库的请求，只在镜像库中进行依赖的搜索以及下载。而如果只是配置了 repository，那么当在 repository 中找不到相应的依赖时，会继续去远程中央库进行搜索和下载。

上面所说的镜像库和 repository 可以通过 <http://www.sonatype.org/nexus/> 进行搭建。

## 项目构建流程

Maven 的构建生命周期中几个常见阶段如下。

- **validate**: 验证项目以及相关信息是否正确。
- **compile**: 编译源代码和相关资源文件。
- **test**: 对测试代码进行测试。
- **package**: 根据不同的项目类型进行打包。
- **verify**: 验证打包的正确性。

- **install**: 将打好的包安装到本地。
- **deploy**: 将打好的包发布到远程库中。

当然, 对应上述每一个阶段, 以及 **pre**、**post**、**proces** 做前缀的一些阶段, 还有一些在命令行中不常用的阶段, 如 **test-compile**、**integration-test** 等。

针对 Java Web 项目, 一个完整的项目构建流程如图 2-4 所示。

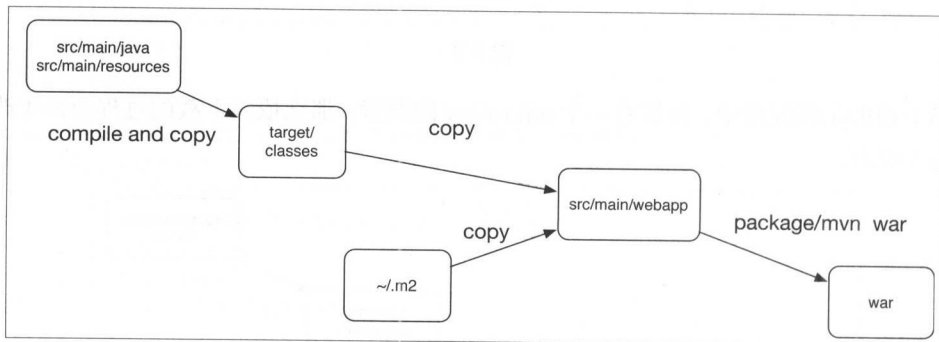


图 2-4

## profile

现实开发中一个很常见的需求就是需要根据不同的环境打包不同的文件。Maven 中的 **profile** 即可解决此问题。

```

<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <resources.dir>src/main/resources/dev</resources.dir>
    </properties>
  </profile>
  <profile>
    <id>test</id>
    <properties>
      <resources.dir>src/main/resources/test</resources.dir>
    </properties>
  </profile>
  <profile>
    <id>prod</id>
    <properties>
      <resources.dir>src/main/resources/prod</resources.dir>
    </properties>
  </profile>
</profiles>
  
```

```

    </profile>
</profiles>

<build>
  <resources>
    <resource>
      <directory>${resources.dir}</directory>
    </resource>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
  </resources>
</build>

```

如此，分为 dev、test 和 prod 共 3 种环境，对应每一种环境，其资源文件路径都不一样。在使用 MVN 时，使用 -P 参数指定 profile 即可生效。

## 复用 test

当需要将测试用例（src/test 下的资源和类）以 jar 包形式发布出去的时候，需要用到 test-jar。首先，在打包时配置 maven-jar-plugin，如下：

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>test-jar</goal>
      </goals>
      <configuration>
        <excludes>
          <exclude>*.conf</exclude>
          <exclude>**/*.*.conf</exclude>
          <exclude>logback.xml</exclude>
        </excludes>
      </configuration>
    </execution>
  </executions>
  <configuration>
    <excludes>
      <exclude>*.conf</exclude>
      <exclude>**/*.*.conf</exclude>
      <exclude>*.properties</exclude>
      <exclude>logback.xml</exclude>
    </excludes>
  </configuration>
</plugin>

```

使用时，指定 dependency 的 type 为 test-jar：

```
<dependency>
  <groupId>xx</groupId>
  <artifactId>xx</artifactId>
  <version>1.0-SNAPSHOT</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
```

## 常用插件

Maven 提供了很多插件方便开发工作。

- **maven-source-plugin**：源码发布插件，绑定在 compile 阶段，执行 jar goal，将源码以 jar 包的形式发布出去。
- **maven-javadoc-plugin**：javadoc 插件，将源码的 javadoc 发布出去。
- **maven-tomcat7-plugin**：此插件可以直接使用 Tomcat 运行 Web 项目，常用的命令是 mvn tomcat7:run。同样的还有 jetty-maven-plugin。
- **maven-shade-plugin**：此插件是 Maven 常用的打包插件，一般将其绑定在 package 阶段，执行其 shade goal。其能够将源码和依赖的第三方资源打包在一起以供独立运行。
- **maven-assembly-plugin**：和 maven-shade-plugin 一样也是打包插件，但是其功能更强大，输出压缩包格式除了 jar 外，还支持 tar、zip、gz 等。
- **maven-gpg-plugin**：此插件是 jar 包的签名插件，可以对自己发布的 jar 包进行签名。

## 提示

- 在项目版本号中加入 SNAPSHOT 后缀作为快照版本，可以使得 Maven 每次都能自动获取最新版本而无须频繁更新版本号。
- mvn -DNAME=test 可以传递给 POM 参数，使用 \${NAME} 引用即可。
- 在 dependency 中设置 optional 为 true，可使得此依赖不传递出去。如下：

```
...
<artifactId>suishen-libs</artifactId>
...

...
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
```

```

<artifactId>httpasyncclient</artifactId>
<version>4.1.3</version>
<optional>true</optional>
</dependency>
...

```

如此一来，依赖于 suishen-lib 的项目除非在自己的 POM 里显示声明，否则不会依赖于 httpasyncclient。

- 由于 Maven 自定义 plugin 很复杂，不够灵活，因此很多时候都是结合 Ant 的灵活性和 Maven 一起使用的。

```

<target name="compile" depends="clean">
  <exec executable="mvn">
    <arg line="compile"/>
  </exec>
</target>

```

```

<target name="compile" depends="clean">
  <exec executable="cmd">
    <arg line="/c mvn compile"/>
  </exec>
</target>

```

- 日常开发中一个工程可能比较庞大，这时可以把这个工程拆分成多个子模块来管理。一个多模块工程包含一个父 POM，在其中定义了它的子模块，每个子模块都是一个独立的工程。

```

<project>
  ...
  <packaging>pom</packaging>

  <modules>
    <module>module-1</module>
    <module>module-2</module>
  </modules>
</project>

```

Maven 常用命令介绍请见附录 A。

### 2.1.3 新兴构建工具——Gradle

Gradle 是目前正在开始流行的新一代构建工具，正在逐步地推广使用，尤其以 Android 为典型。



## 介绍

基本上现在所有的 Android 项目都采用 Gradle 作为项目构建工具。概括来讲，有以下几个特点。

- 采用了 Groovy DSL 来定义配置，相比 XML 更加易于学习和使用，并大大简化了构建代码的行数。此外，这种“配置即代码”的方式能够大大简化配置学习和插件编写的成本，提供了更好的灵活性。
- 在构建模型上非常灵活。可以轻松创建一个 Task，并随时通过 depends 语法建立与已有 Task 的依赖关系。这里 Gradle 使用了 Java 插件来支持 Java 项目的标准构建生命周期（和 Maven 类似）。
- 依赖的 scope 简化为 compile、runtime、testCompile、testRuntime 共 4 种。
- 支持动态的版本依赖。在版本号后面使用 + 号的方式可以实现动态的版本管理。
- 支持排除传递性依赖或者干脆关闭传递性依赖。
- 完全支持 Maven、Ivy 的资源库（repository）。

## 使用示例

Gradle 的配置写在 build.gradle 文件中：

```

apply plugin: 'groovy'
apply plugin: 'idea'
apply plugin: 'checkstyle'

// --- properties ---
ext.ideaInstallationPath = '/Applications/IntelliJ IDEA.app/Contents'
sourceCompatibility = 1.6
// --- properties ---

// 源码目录结构
sourceSets.main.java.srcDirs = []
sourceSets.main.groovy.srcDir 'src/main/java'

// 增加 repository
repositories {
    mavenLocal()
    maven {
        url "http://maven.etouch.cn/nexus/content/groups/public/"
    }
    mavenCentral()
}

// 依赖管理

```

```
dependencies {
    compile fileTree(dir: ideaInstallationPath + '/lib', include:
        '*.jar')
    testCompile 'org.mockito:mockito-core:2.0.3-beta'
    testCompile 'org.assertj:assertj-core:1.7.1'
    testCompile 'org.springframework:spring-test:4.0.0.RELEASE'

    // 排除全部或特定的间接依赖
    runtime ('commons-dbcp:commons-dbcp:1.4') {
        transitive = false
        // 或 exclude group: xxx, module: xxx
    }

    // 然后显式声明
    runtime 'commons-pool:commons-pool:1.6'
}

//Gradle Wrapper
task wrapper(type: Wrapper) {
    gradleVersion = '3.0'
}

task helloWorld

helloWorld << {
    println "Hello World!"
}

task testA(dependsOn:helloWorld)

testA << {
    println "test"
}

task copyFile(type: Copy)
//task(copyFile(type: Copy))

copyFile {
    from 'xml'
    into 'destination'
}
```

可见，依赖（dependency）的配置相比 Maven 得到了大大的简化，对于任务（Task）的定义也非常简单。

## 提示

- `gradle -Penv=test` 可以传递参数，使用 `env` 引用即可。这里需要注意的是，Gradle 中默认并没有提供 Maven 的 profile 支持，但是可以利用 `-P` 参数自己实现此功能。

- Gradle 中配置的语法和平常所见的 Groovy 非常不同，其利用了 Groovy 的 AST 转换等特性实现了自己的一套语法。
- 建议使用 Gradle Wrapper (`gradle wrapper --gradle-version 3.0`) 来做 Gradle 操作。一方面可以使得项目成员不用预先安装 Gradle，另一方面还便于统一项目所使用的 Gradle 版本。
- 务必要保持构建脚本简洁、清晰，如把属性、常量(如版本号)放到 `gradle.properties` 中。
- 模块化构建脚本，通过 plugin 机制在多个项目中复用，共享相关的配置 `apply from: <link_to_gradle>`。通过这种方式可以统一团队或者公司的一些构建规范、依赖版本等。

Gradle 的常用命令介绍请见附录 A。

## 2.2 代码版本控制

在平时的开发过程中，尤其是当多人协同开发时，经常会遇到以下问题。

- 代码管理混乱。
- 代码冲突。
- 在代码整合期间引发 Bug。
- 无法对代码的拥有者进行权限控制。
- 项目不同版本的发布较困难。

代码版本控制 (Version Control) 通过追踪文件的变化解决上述问题。

目前常用的版本控制系统 (VCS) 包括 SVN 和 Git。

### 2.2.1 集中式代码版本管理——SVN

SVN 是前几年用得最普遍的一个 VCS 工具，采用了分支管理系统，具有以下几个特点。

- 原子提交。一次提交不管是单个还是多个文件，都是作为一个整体提交的。在这当中发生的意外，例如传输中断，不会引起数据库的不完整和数据损坏。
- 重命名、复制、删除文件等动作都保存在版本历史记录中。对于二进制文件，使用了节省空间的保存方法 (只保存和上一版本不同之处)。

- 目录也有版本历史。整个目录树可以被移动或者复制，操作很简单，而且能够保留全部版本记录。
- 分支的开销非常小。
- 优化过的数据库访问，使得一些操作不必访问数据库就可以做到。这样减少了很多不必要的和数据库主机之间的网络流量。
- 集中式版本控制，依赖于中央版本服务器。

## 流程与常用命令

一个通常较简单的 SVN 工作流程如图 2-5 所示。

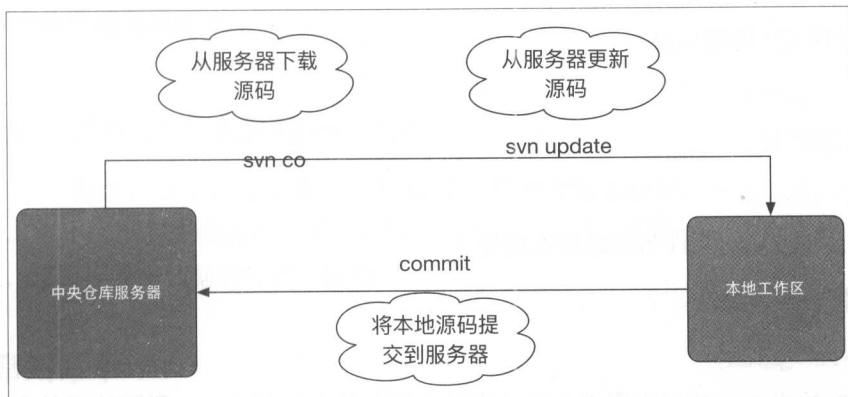


图 2-5

1) 在加入一个新项目的开发中时，从项目 SVN 地址获取代码。

```
svn co [svn_url]
```

2) 把改变的文件添加到版本库中。

```
svn add [file]
```

这里使用 `svn add -a` 可以将所有变动都添加到版本库中。

3) 将改动的文件提交到版本库中。

```
svn commit -m [LogMessage]
```

4) 更新到某个版本。

```
svn update -r [version] [path]
```

如果 `svn update` 后面没有目录，默认更新当前目录以及子目录下的所有文件，如果没有版本号，则更新到最新版本。

5) 查看文件或者目录状态。

```
svn status [path]
```

对应于每一个文件都有这几个状态: [?: 不在SVN的控制中; !: 被删除; M: 内容被修改; C: 发生冲突; A: 预定加入到版本库; K: 被锁定 ]。

6) 删除文件。

```
svn delete path -m "delete test file "  
svn delete [file]  
svn ci -m 'xxxx'
```

7) 查看日志。

```
svn log [path]
```

8) 查看文件详细信息。

```
svn info [path]
```

9) 比较差异。

```
svn diff [path]
```

这里是将改动的文件与基础版本比较。

10) 恢复本地修改。

```
svn revert [path]
```

恢复原始未改变的工作副本文件(恢复大部分的本地修改), 需要注意的是此命令不会恢复被删除的目录。

11) 代码库 URL 变更。

```
svn switch [svn_url]
```

switch 仅限于同一个 repository 下的目录之间, switch 并不关心切换的目标分支与源分支之间的关系。switch 的代价很小, 因此应尽量使用 switch, 而不是完整的 check out。

12) 解决冲突。

```
svn resolved [path]
```

移除工作副本的目录或文件的“冲突”状态。这里只是移除冲突的相关文件, 然后让 PATH 可以再次提交。

## 服务器搭建

SVN 是集中式版本控制系统, 依赖于中央版本服务器。Windows 下使用 VisualSVN Server 即可, Linux 下通过官网的源码包编译安装即可, 不再详述。

## 提示

- 应尽量避免的操作：
  - 格式化代码，完全改变文件的行号缩进等。
  - 直接用另一个版本的文件覆盖当前文件。
  - 在 Windows 下更改文件名，但只改了文件名的大小写。
- 保证提交的完整性：零碎提交功能不完整的代码将引起协同开发的同事以及 QA 很难运行代码，且不利于版本的回退等操作。
- 不应提交的文件：以“.”开头的文件、目录，例如 .classpath、缩略图文件、class 文件。
- 提交代码前务必先 `svn update`。

## 2.2.2 分布式代码版本管理——Git

Git 是一个分布式版本控制系统，相比 SVN 的集中式版本控制，每个工作计算机都可以作为版本库，可以不依赖远程仓库服务器离线工作。此外，它的每一个版本库都保存了完整信息，且是按照元数据的方式存储的。

### 流程与常用命令

如图 2-6 所示，使用 Git 的流程一般如此，通常使用图中的 6 个命令即可。

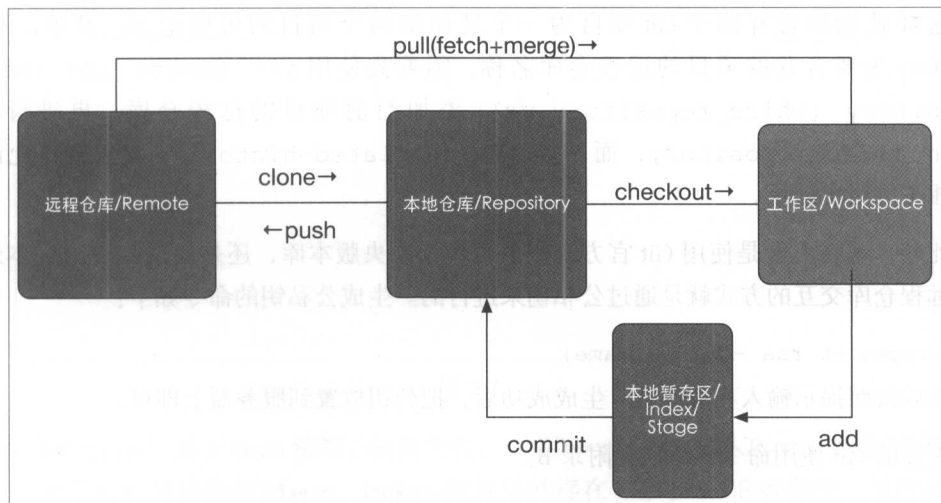


图 2-6

1) 复制项目的 Git 地址。

```
git clone [repository url]
```

2) 切换项目分支。

```
git checkout [branchName]
```

3) 增加或改动了一些文件。

```
git add [fileName]
```

这里也可以使用 `git add -a` 添加所有变动到暂存区。

4) 提交文件到本地仓库。

```
git commit -m <message>
```

可以使用 `git commit -a` 跳过 `git add` 步骤直接 `commit`。

5) 提交到远程仓库。

```
git push origin master
```

6) 从远程库更新代码并合并。

```
git pull
```

此命令是 `git fetch` 和 `git merge` 的结合。

7) 合并不相关的分支。

```
git merge [third_repository]/master --allow-unrelated-histories
```

这样就能够合并两个 Git 项目为一个且保留两个项目的历史记录。其中, `third_repository` 为要合并的项目的远程仓库名称, 需要先使用 `git remote add third_repository [third_repository_url]` 添加当前项目的远程仓库, 再进行 `git fetch third_repository`, 而 `--allow-unrelated-histories` 参数则是允许合并不相关的历史记录。

此外, 现在不管是使用 Git 官方的服务器作为中央版本库, 还是使用 GitHub, 本地仓库与远程仓库交互的方式就是通过公私钥来进行的。生成公私钥的命令如下:

```
ssh-keygen -t rsa -C [userName]
```

然后按照提示输入相应信息, 生成成功后, 把公钥放置到服务器上即可。

更多的 Git 使用命令介绍请见附录 B。

## 服务器搭建

虽然 Git 是分布式的，并不需要中央版本服务器即可使用，但是当多人协作时，中央版本服务器必不可少。

按照官方文档搭建一个 Git 服务器是比较烦琐的。面对这些复杂的操作步骤以及 Git 本身上手的门槛，很多人望而却步，造成了之前 Git 并没有那么普及的状况。而一切从 GitHub 横空出世开始改变了，这个看似简单的网站给很多人带来了简单、极致、方便的代码托管服务，同时也给全世界带来了一股开源的风潮。很多初创公司或者小公司直接选择使用 GitHub 建立自己的代码仓库。

与此同时，许多 GitHub 的开源实现层出不穷，如今你只需要下载一个类似的实现部署到服务器上，就能拥有自己的“GitHub”。

目前比较知名、用得较多的 GitHub 实现，有以下几个。

- **GitLab**: Ruby on Rails 实现，是目前最出名也最强大的 GitHub 克隆实现，并且除了版本管理外，集成了项目管理、持续集成等很多功能。官网提供了很多操作系统下的一键安装包。遗憾的是，GitLab 现在已经商业化，一些强大的功能只有在其收费版本中才能体验到。其界面如图 2-7 所示。

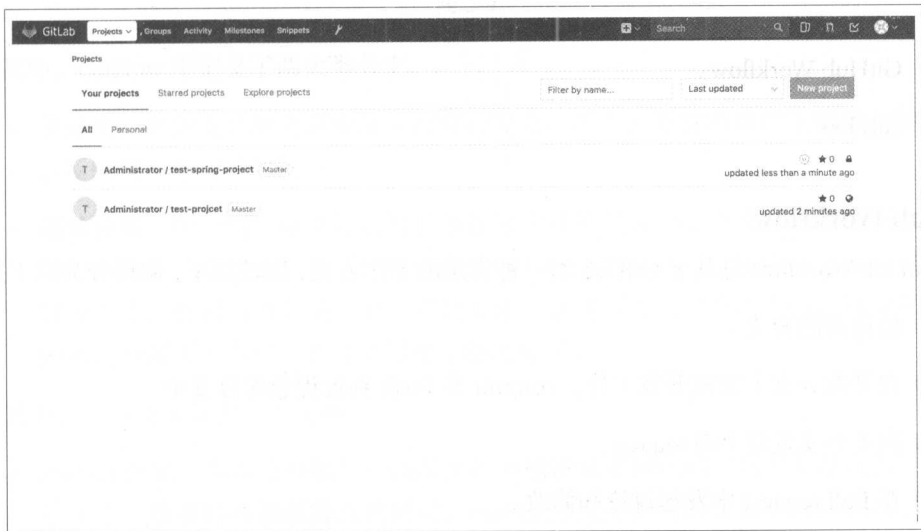


图 2-7

- **GitBucket**: 基于 Scala 编写，极易安装，“扔”一个 war 包到 Tomcat 就能完成部署，完全可以和其他如 Maven、Jenkins 的软件并存在一个 Java EE 容器中。虽然功能没有 GitLab 那么强大，但胜在简单。更重要的是，对于 Java 工程师来说是友好的，很方便做二次开发、拓展。其界面如图 2-8 所示。



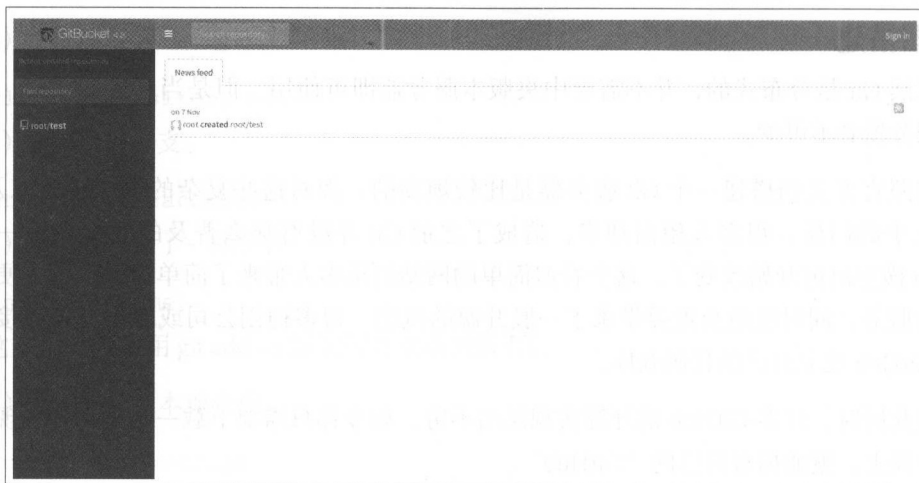


图 2-8

## 工作流

由于 Git 是分布式版本控制系统，虽然带来了许多优势，但是同时带来了协同工作的复杂性。因此需要一套基于 Git 的工作流来规范整个协同流程。目前，比较流行的有以下两种。

- GitHub Workflow。
- GitFlow。

### GitHub Workflow

GitHub Workflow 是基于 GitHub 的一种常用的工作方式，比较简单，流程分为以下几步。

- 1) 检出新的分支。
- 2) 在开发分支上完成开发工作，commit 并 Push 到远程仓库分支中。
- 3) 向主分支发起 Pull request。
- 4) 在 Pull request 中发起讨论和修改。
- 5) 将开发分支部署到测试环境，经测试无误后 merge 回主分支。

这里最核心的就是基于 Pull request 的协作方式，在类似于 GitHub 的这种应用中，还可以基于此来进行 code review、任务沟通等工作。

## GitFlow

相比 GitHub Workflow, GitFlow 根据 Git 原来的命令和语义, 做了一层语义抽象, 多了很多新的定义, 从源代码管理的角度来说, 对通常意义上的软件开发活动进行了约束, 流程如图 2-9 所示。

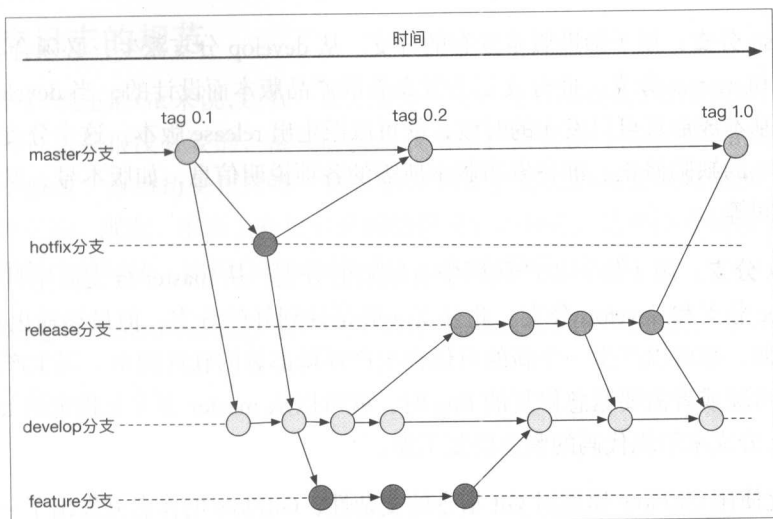


图 2-9

其中, GitFlow 中定义了两大类分支。

- **主分支:** 主分支是所有开发活动的核心分支。所有的开发活动产生的输出物最终都会反映到主分支的代码中。
- **辅助分支:** 用于组织解决特定问题的各种软件开发活动的分支。辅助分支主要用于组织软件新功能的并行开发、简化新功能开发代码的追踪、辅助完成版本发布工作以及对生产代码的缺陷进行紧急修复工作。这些分支与主分支不同, 通常只会在有限的时间范围内存在。其也可以视作临时分支。

其中, 主分支又分为以下 2 种。

- **master 分支:** 存放的是随时可供在生产环境中部署的代码。当开发活动告一段落, 产生了一份新的可供部署的代码时, master 分支上的代码会被更新。同时, 每一次更新, 都添加对应的版本号标签 (tag)。
- **develop 分支:** 保存当前最新开发成果的分支。通常这个分支上的代码也是可进行每日 / 夜间发布的代码。当 develop 分支上的代码已实现了软件需求说明书中所有的功能, 通过了所有的测试, 并且代码已经足够稳定时, 就可以将所有的开发成果合并回 master 分支了。

辅助分支又分为以下 3 种。

- **feature 分支**：用于开发新功能时所使用的分支。从 develop 分支发起 feature 分支，代码必须合并回 develop 分支。此分支甚至可以仅仅保存在开发者自己的代码库里而不提交。
- **release 分支**：用于辅助版本发布的分支。从 develop 分支派生，必须合并回 develop 分支和 master 分支。此分支是为发布新的产品版本而设计的，当 develop 上开发的功能基本成形且可以发布的时候，就可以派生出 release 版本。这个分支上的代码允许做小的缺陷修正、准备发布版本所需的各项说明信息，如版本号、发布时间、编译时间等。
- **hotfix 分支**：用于修正生产代码中有缺陷的分支。从 master 分支派生且必须合并回 master 分支和 develop 分支。此分支一般是计划外的分支，但最终输出和 release 分支类似，都可以产生一个新的可供在生产环境部署的软件版本。当生产环境遇到了异常情况或者需要紧急修复的 Bug 时，就可以从 master 分支上指定的 tag 版本派生 hotfix 分支来组织代码的紧急修复工作。

为了简化使用 GitFlow 模型时 Git 指令的复杂性，GitFlow 的作者开发出了一套 Git 增强指令集，可以运行于 Windows、Linux、UNIX 和 Mac 操作系统之下，地址见 <https://github.com/nvie/gitflow>。

## 提示

- .gitignore 中添加不需要版本管理的文件，如 .DS\_Store logs target、node\_modules 等。根据工程的类型不同这里需要加入的文件也不同。
- git push 之前需要先 git pull 更新代码。
- 如果想要忽略已经在版本库里的文件 / 文件夹，即有一个版本库里的文件，你做了改动但并不想提交到版本库中，这时可以使用命令 `git update-index --assume-unchanged <file>`，之后可以通过 `git update-index --no-assume-unchanged <file>` 恢复追踪。通过 `git ls-files -v | grep -e "^[hsmrck]"` 可以列出当前被忽略的、已经纳入版本库管理的文件。
- 使用 `git cherry-pick <commit id>` 可以选择某一个分支中的一个或几个 commit(s) 来进行操作。

- 永远不要在所有人都在的公共开发分支上做 rebase 操作。一般情况下在临时分支上是需要 rebase 主分支代码的，而 merge 则主要用在主分支上将临时分支的代码合并过来，然后就可以删除临时分支了。

### 2.2.3 提交日志的规范

不论对于 SVN 还是 Git 来说，还有一点尤为重要的就是每次提交到代码库时的日志撰写。很多人都认为日志是很没必要的，浪费时间还没啥用，其实撰写清晰规范的格式化日志有助于追踪版本修改、查看历史记录等。SVN 的默认配置是允许提交空日志的，但 Git 却是不允许日志为空的。现在，市面上有很多类似的提交日志规范，这里推荐使用 angular 规范，它是目前使用最广的写法，比较合理和系统化，并且有配套的工具。

angular 规范的 commit message 包括 3 个部分 header、body 和 footer，格式如下：

```
<type>(<scope>): <subject>
// 空一行
<body>
// 空一行
<footer>
```

- type 用于说明 commit 的类别，只允许使用下面 7 个标识。
  - feat: 新功能 (feature)。
  - fix: 修补 Bug。
  - docs: 文档 (documentation)。
  - style: 格式 (不影响代码运行的变动)。
  - refactor: 重构 (既不是新增功能，也不是修改 Bug 的代码变动)。
  - test: 增加测试。
  - chore: 构建过程或辅助工具的变动。
- scope 用于说明 commit 影响的范围，比如数据层、控制层、视图层等，视项目不同而不同。
- subject 是 commit 目的的简短描述，不超过 50 个字符。
- body 部分是对本次 commit 的详细描述，可以分成多行。

- footer 部分只用于两种情况：不兼容变动时，以 BREAKING CHANGE 开头，后面是对变动的描述以及变动理由和迁移方法；如果当前 commit 针对某个 issue，那么可以在 footer 部分关闭这个 issue。
- 还有一种特殊情况，如果当前 commit 用于撤销以前的 commit，则必须以 revert: 开头，后面跟着被撤销 commit 的 header。body 部分的格式是固定的，必须写成 This reverts commit <hash>., 其中的 hash 是被撤销 commit 的 SHA 标识符。

一个符合 angular 规范的 commit message 示例如图 2-10 所示。

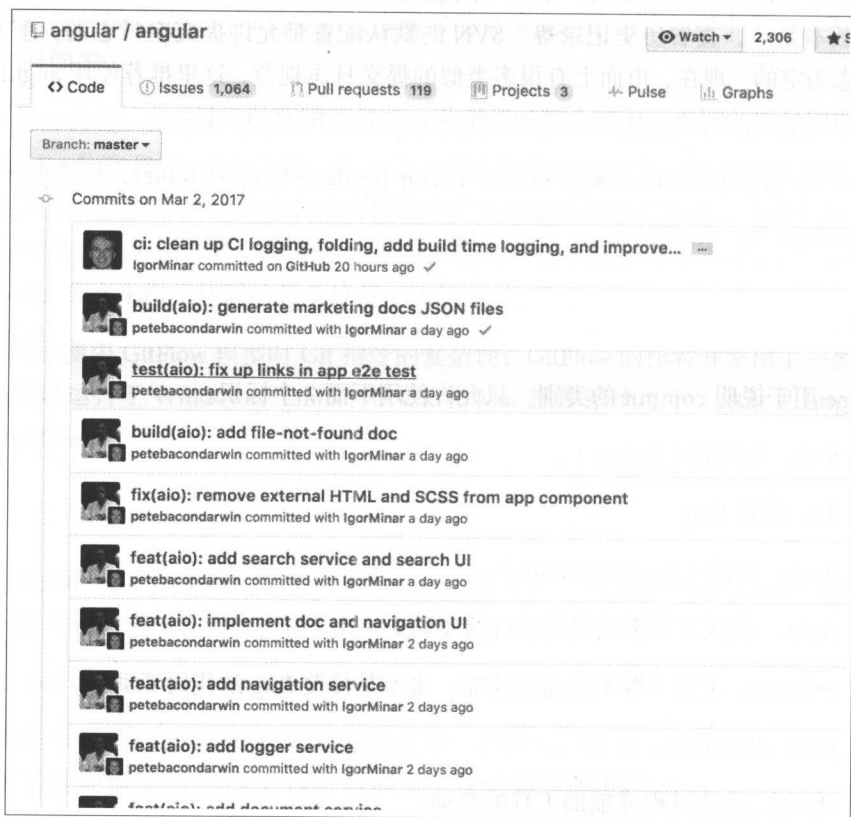


图 2-10

如此规范 commit message 可以带来以下好处。

- 提供更多的历史信息，方便快速浏览。可以对代码的历史记录一目了然，能够大体上知道每次提交都做了什么。如下：

```
git log [last tag] HEAD --pretty=format:%s
```

历史信息如图 2-11 所示。

```

ci: clean up CI logging, folding, add build time logging, and improve error handling (#14425)
build(aio): generate marketing docs JSON files
test(aio): fix up links in app e2e test
build(aio): add file-not-found doc
fix(aio): remove external HTML and SCSS from app component
feat(aio): add search service and search UI
feat(aio): implement doc and navigation UI
feat(aio): add navigation service
feat(aio): add logger service
feat(aio): add document service
feat(aio): add location service
build(aio): content documents should have a `contents` field
docs(aio): add NOTES
fix(aio): remove previous files
build: update to yarn 0.21.3 (#14805)
docs: add 2.4.9 release notes
release: cut the 4.0.0-rc.2 release
docs: add changelog for 4.0.0-rc.2
docs: update 2.4.9 release schedule
fix(animations): make animations work in AOT (#14775)
build: fix secondary entry point es5 output and core Rx references (#14820)
docs: fix changelog to list npm command for Windows (#14812)

```

图 2-11

- 可以过滤某些 commit（比如文档改动），便于快速查找信息，比如要查看新增加的功能。如下：

```
git log [last release] HEAD --grep feat
```

查看新增加的功能如图 2-12 所示。

```

commit dca83ec7389d969d9417dcbfe1013ba5439a797e
Author: Peter Bacon Darwin <pete@bacondarwin.com>
Date:   Wed Mar 1 23:05:16 2017 +0000

    feat(aio): add search service and search UI

commit 71e22b8d11994841206aa5402a738f2f69aadbe4
Author: Peter Bacon Darwin <pete@bacondarwin.com>
Date:   Wed Mar 1 14:30:25 2017 +0000

    feat(aio): implement doc and navigation UI

commit 371dc4744cdc2bd04d5433611f26eca463adb0d2
Author: Peter Bacon Darwin <pete@bacondarwin.com>
Date:   Wed Mar 1 11:55:46 2017 +0000

    feat(aio): add navigation service

commit 4767f107fbc5a6c8fa6f7723bae76e2a05df18c1
Author: Peter Bacon Darwin <pete@bacondarwin.com>
Date:   Wed Mar 1 11:27:54 2017 +0000

    feat(aio): add logger service

```

图 2-12

- 使用 conventional-changelog 可以直接从 commit 生成 change Log（在发布新版本时，用来说明与上一个版本差异的文档）。如下：

```
conventional-changelog -p angular -i CHANGELOG.md -w
```

change Log 如图 2-13 所示。

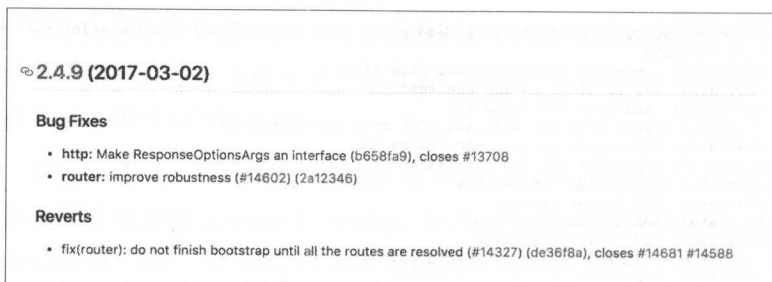


图 2-13

此外，可以使用 Commitizen 来撰写符合规范的 commit message。

虽然以上介绍使用 Git 来做示例，但是对于 SVN 也是可以按照此套规范来进行的。

## 2.3 代码质量保证

代码质量是任何技术团队都非常重视的。一个具有良好代码质量的项目，不仅能够保证应用的稳定、高效运行，也能够大大降低后续的维护成本，便于重构、扩展等。因此，代码质量也是 Java 项目工程化至关重要的一部分。

我们可以将代码质量分成以下几个等级。

- 可编译。
- 可运行。
- 可测试。
- 可读。
- 可维护。
- 可重用。

其中，可编译、可运行是完成一个项目最基本的目标。至于可维护和可重用，目前并没有非常好的工程化工具来保障。很多时候需要工程师的个人素养和团队的监督机制。因此，可测试、可读是我们至少要达到的目标。可测试性即保证你的代码能够经受住各种黑盒和白盒测试而不会出现 Bug，毕竟上线的代码面临的外部输入是不可预测的。而可读性则显得更重要，毕竟代码是用来给人阅读的。而什么是好的可读性呢？当你自己写的代码过了 1 个月、3 个月再看，能很快说出自己的编程逻辑并定位某个功能的代码位置，这就是一种可读的表现。

本章主要介绍代码质量达到可测试、可读可以依靠的一些工程化工具。

### 2.3.1 使用单元测试保证代码质量

单元测试, 即 Unit Test, 指的是对代码的各个接口的测试。所谓的“测试驱动开发”也依赖于此。单元测试, 可以在某种程度上保证代码的稳定性, 并且能够保证在做代码变更、迭代时原有代码逻辑的正确性。

#### JUnit

Java 中一般使用 JUnit 作为单元测试框架。以下是一个使用 JUnit 做单元测试的例子:

```
public class UtilTest {
    @BeforeClass
    public static void initClass(){
        System.out.println("i will be called only once,before the first
test method");
    }

    @AfterClass
    public static void afterClass(){
        System.out.println("i will be called only once,after the last
test method");
    }

    @Before
    public static void initMethod(){
        System.out.println("i will be called before every test method");
    }

    @After
    public static void afterMethod(){
        System.out.println("i will be called after every test method");
    }

    @Test
    public void testAdd() throws Exception {
        Assert.assertEquals(2, add(1,1));
    }

    @Test(timeout=100)
    public void testTimeout(){
        try{
            Thread.sleep(500);
        }catch(InterruptedException e){

        }
    }

    @Test(expected=IndexOutOfBoundsException.class)
    public void testException(){
```



```

        new LinkedList<String>().get(0);
    }

    @Ignore("ignore the test")
    @Test
    public void ignoreTest() throws Exception {
        Assert.assertEquals(2, add(1,1));
    }

    public int add(int n1, int n2){
        return n1 + n2;
    }
}

```

这个例子很简单，就是验证 add 方法的正确性。

这里使用了 @Test 注解来声明一个单元测试。可以通过 Maven 来调用这个测试：

```
mvn test -Dtest=*.UtilTest
```

在 Maven 的构建周期中，默认执行所有位于 src/test 下的测试用例。

如例子所示，JUnit 中还有以下几个常用注解。

- 在某些阶段必然被调用的代码。
  - Class 级（用在 static 方法）。
    - ◆ @BeforeClass 在第一个测试方法运行前执行。
    - ◆ @AfterClass 在最后一个测试方法运行后执行。
  - Method 级（用在实例方法）。
    - ◆ @Before 在每个测试方法执行前执行。
    - ◆ @After 在每个测试方法执行后执行。
- 对运行时间有要求的测试，测试运行时间超时则失败。
 

```
@Test(timeout=expire-milliseconds)
```
- 期待抛出异常的测试用例，测试方法中如果不抛出异常则失败。
 

```
@Test(expected=ExpectedException.class)
```
- 忽略掉不常用或者未完成的测试用例。
 

```
@Ignore("some message")
```
- 使用 @Suite.SuiteClasses 将一组测试用例打包运行。

```

@Suite.SuiteClasses({
    UtilTest.class,
    HelloTest.class
})
public class SuiteTest {

}

```

JUnit 还有一种比较高级的参数化测试，即需要多组参数验证的测试用例，如下：

```

@RunWith(Parameterized.class)
public class ParameterizedTest {
    private int param;
    private boolean result;

    // 为每组数据构建测试用例
    public ParameterizedTest(int param, boolean result) {
        this.param = param;
        this.result = result;
    }

    // 生成测试数据
    @Parameterized.Parameters
    public static Collection<Object[]> genParams() {
        return Arrays.asList(new Object[][]{{1, true}, {2, false}});
    }

    // 测试代码
    @Test
    public void test() {
        Assert.assertEquals(this.param % 2 == 1, this.result);
    }
}

```

此外，在对使用了 Spring 的项目写测试用例的时候，需要依赖 SpringJUnit4ClassRunner（需要引入 spring-test 依赖）：

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
    "classpath:applicationContext.xml",
})
public class SpringTest {
    @Resource
    private UserDao userDao;

    @Test
    public void testAdd(){
        User u = userDao.add(new User("张三"));

        Assert.assertTrue(u.getId() > 0);
    }
}

```

## Mock 测试

在写单元测试时会经常碰到一些依赖于外部环境的场景,比如数据库查询、网络请求等。有时候,这些外部依赖并非是一直可用的或者暂时没有开发完成的,经常会对测试产生影响。而 Mock 测试工具则是为了解决这个问题开发的。Java 中常用的 Mock 测试框架主要是 EasyMock 和 PowerMock。这里的 Mock 就是打桩(Stub)或者模拟,当你调用一个不好在测试中创建的对象时,Mock 框架为你模拟一个和真实对象类似的替身来完成相应的行为。

以 EasyMock 的使用举例,来看看如何使用 Mock 框架写单元测试。

有一个 AppService 的接口实现,依赖于 AppDao 接口:

```
public AppServiceImpl implements IAppService{
    private AppDao appDao;

    public App getAppById(long id){
        return appDao.getById(id);
    }

    public void updateAppById(long id){
        return appDao.update(app);
    }

    public void setAppDao(AppDao appDao){
        this.appDao = appDao;
    }
}
```

如果这时候 AppDao 并没有实现,那么若对 AppService 做单元测试,就需要 Mock 出一个 AppDao:

```
public class AppServiceImplTest {
    @Test
    public void testGet() {
        App expectedApp= new App();
        app.setId(1);
        AppDao mock = EasyMock.createMock(AppDao.class); // 创建 Mock 对象
        EasyMock.expect(mock.getById(1)).andReturn(expectedApp);
        // 录制 Mock 对象预期行为
        EasyMock.replay(mock);
        // 重放 Mock 对象,测试时以录制的对象预期行为代替真实对象的行为

        AppServiceImpl service = new AppServiceImpl();
        service.setAppDao(mock);
        App app = service.getAppById(1); // 调用测试方法
        assertEquals(expectedApp, app); // 断言测试结果
    }
}
```

```

    EasyMock.verify(mock); // 验证 Mock 对象被调用
}
}

```

这里需要注意的是,在 EasyMock 3.0 之前,其使用 JDK 的动态代理实现 Mock 对象创建,因此只能针对接口进行 Mock。3.0 版本后则加入了使用 CGLIB 的实现方式,因此支持对普通类的 Mock。

以上是一个简单的测试,除此之外,还有以下常用场景。

- 基本参数匹配,即对所有匹配的参数都做 Mock。

```

EasyMock.expect(mock.getById(EasyMock.isA(Long.class))).
    andReturn(exceptedApp);
// 这样无论传入的 ID 是多少,都会返回这个 expectedApp 而不是发生 Unexpected
// method 的 Mock 异常。

```

类似于 EasyMock.isA 的还有 anyInt()、anyObject()、isNull()、same()、startsWith() 等。

- AppServiceImpl 在调用 AppDao 的 getById 方法时发生异常。

```

EasyMock.expect(mock.getById(1)).andThrow(new RuntimeException());

```

- void 方法 Mock。

```

mock.updateAppById(1);
EasyMock.expectLastCall().anyTimes();

```

使用 expectLastCall 方法即可。

- 多次调用返回不同值的 Mock。

这种业务场景的例子就是 iterator 的 hasNext 方法,如果一直返回 true,那么代码会陷入无限循环。正常的测试逻辑应该是先返回几次 true 执行循环体,然后返回 false 退出循环。如下:

```

EasyMock.expect(rs.next()).andReturn(true).times(2).
    andReturn(false).times(1);

```

使用 EasyMock 已经可以解决大部分测试需求,但由于 EasyMock 使用 JDK 动态代理以及 CGLIB 实现子类的方式,其对于静态、final 类型的类和方法以及私有方法是无法完成 Mock 的。PowerMock 则在 EasyMock 的基础上进行了扩展,使用字节码操作技术直接对生成的字节码类文件进行修改,从而能够对静态、final 类型的类和方法、私有方法进行 Mock,还可以对类进行部分 Mock。

PowerMock 的工作过程和 EasyMock 类似,不同之处在于需要在类层次声明 @RunWith(PowerMockRunner.class) 注解,以确保使用 PowerMock 框架引擎执行单元测试。

- 静态方法。

```
// Util.java
public class Util{
    public static String getEnv(String name){
        return System.getenv(name);
    }
}

//UtilTest.java
@RunWith(PowerMockRunner.class)
@PrepareForTest({Util.class})// 声明要Mock 的类
public class UtilTest {
    @Test
    public void staticMethodMockTest() throws Exception {
        PowerMock.mockStatic(System.class);//Mock 静态方法
        EasyMock.expect(System.getenv("java_home")).andReturn("env
value");// 录制 Mock 对象的静态方法
        PowerMock.replayAll();// 重放 Mock 对象
        Assert.assertEquals("env value",Util.getEnv("java_home"));
        PowerMock.verifyAll();// 验证 Mock 对象
    }
}
```

静态的 final 方法也是如此。这里需要注意的是，对于 JDK 的类，如果要进行静态或 final 方法 Mock，@PrepareForTest() 注解中只能放被测试的类，而非 JDK 的类，如上面例子中的 UtilTest.class。对于非 JDK 的类，如果需要进行静态 final 方法 Mock，@PrepareForTest() 注解中直接放方法所在的类，如果上面例子中的 System 不是 JDK 的类，则可以直接放 System.class。@PrepareForTest({……}) 注解既可以加在类层次上（对整个测试文件有效），也可以加在测试方法上（只对测试方法有效）。

- 非静态的 final 方法。

和静态方法一样。对于非静态方法，如下：

```
public class FinalClass {
    public final String finalMethod() {
        return "final method";
    }
}

public class FinalUtil{
    public String callFinalMethod(FinalClass cls) {
        return cls.getName();
    }
}

@RunWith(PowerMockRunner.class)
public class FinalMethodMockTest {
```

```

@Test
@PrepareForTest(FinalClass.class)
public void testCallFinalMethod() {
    FinalClass cls = PowerMock.createMock(FinalClass.class);
    // 创建 Mock 对象
    FinalUtil util = new FinalUtil();
    EasyMock.expect(cls.finalMethod()).andReturn("cls");
    PowerMock.replayAll();
    Assert.assertTrue(util.callFinalMethod(cls));
    PowerMock.verifyAll();
}
}

```

- 私有方法。

```

public class PrivateMockClass
    private boolean privateMethod(final String name) {
        return true;
    }
}

@RunWith(PowerMockRunner.class)
@PrepareForTest(PrivateMockClass.class)
public class PrivatMethodMockTest {
    @Test
    public void testPrivateMock() throws Exception {
        PrivateMockClass mock = PowerMock.createPartialMock(Private
MockClass.class, "privateMethod");// 只对 privateMethod 方法 Mock
        PowerMock.expectPrivate(mock, "privateMethod", "name").
andReturn(true);// 录制
        PowerMock.replay(mock);
        assertTrue(mock.privateMethod("name"));
        PowerMock.verify(mock);
    }
}

```

这种方式除了私有方法，还可以用在被测试方法都在同一个类且不容易创建的场景下。

除了上述场景外，经常会遇到在测试方法中调用构造函数产生对象。那么如何对构造方法进行 Mock 呢？如下：

```

File mock = createMockAndExpectNew(File.class, "mockPath");
EasyMock.expect(mock.exists()).andReturn(false);
EasyMock.expect(mock.mkdirs()).andReturn(true);
PowerMock.replay(mock, File.class);

```

...

```

PowerMock.verify(mock, File.class);

```

以上代码即可完成对 File 的构造方法的 Mock（对于 mockPath 的 File，其 exists 和 mkdirs 方法都会返回 true）。

## 2.3.2 衡量单元测试的标准

有了单元测试，那么就需要有一个能够衡量单元测试好坏的标准，即测试覆盖率。即你写的单元测试对你的项目中代码的覆盖程度。对于很多比较严格的企业来说，对测试覆盖率的要求经常是达到 98% 以上。当然，在进行测试覆盖率统计的时候，会忽略掉诸如 getter、setter 这些根本没必要进行单元测试的方法。

目前常用的测试覆盖率工具有 EMMA、Cobertura 等。使用方法基本类似，可以使用 IDE 的相关插件，也可以使用相应的 Maven 插件。但相比 Cobertura，EMMA 的覆盖率更为严格，并且支持 test 代码和 Java 代码不在一个工程下的单元测试覆盖率统计。

在 Maven 项目下，直接执行 maven emma:emma 命令即可在 target/site 下生成报告 index.html，示例如下：

```
public class MyClass {
    public void testMethod(int a) {
        if (a % 2 == 0) {
            System.out.println("even");
        } else {
            System.out.println("odd");
        }

        System.out.println(a % 2 != 0 ? "odd" : "even");
    }
}

public class MyClassTest {
    private static MyClass myClass;

    @BeforeClass
    public static void init() {
        myClass = new MyClass();
    }

    @Test
    public void testMethod() {
        myClass.testMethod(1);
    }
}
```

如图 2-14 所示，EMMA 有多种级别的覆盖率指标：类、方法、语句块（basic block）和行，可以根据不同的情况选取不同的指标作为标准。图 2-14 中下面部分是关于具体的某一个类

中的覆盖信息，其中绿色表示覆盖到的代码，红色表示未覆盖到的代码，黄色则表示部分覆盖到的代码。

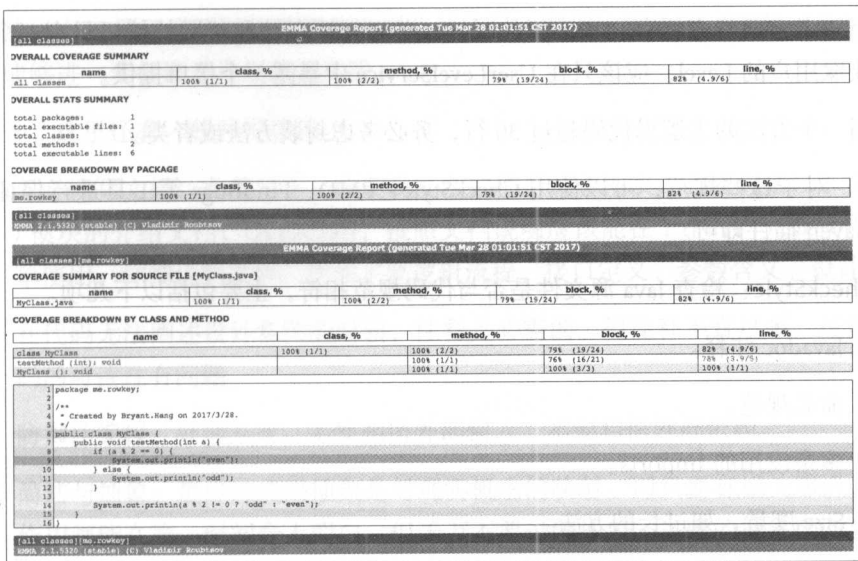


图 2-14

### 2.3.3 开发规范与建议

代码规范是使代码达到可读的关键。由于 Java 语言在发明的时候并没有考虑到语法和规范的统一，因此在编写 Java 代码或者架构 Java 项目时，很多人都有自己一套个性的风格。对于一个团队来说，这并不可取。于是，很多大公司慢慢演化出了自己的编码规范，比较出名的有 Google 和阿里的 Java 编码规范。

这里根据笔者自己的一些经验和实践，列出一些最需要遵循的 Java 编码规范。

- 变量命名按照 Java 通用方式 Camel 命名法。常量尤其注意使用全大写\_（下划线连接）单词的方式，如 `USER_KEY`。
- 变量和类命名务必具有意义，能让人一眼看出表示的意思。如 `userList` 表示用户列表，不要使用 `list`、`set`、`button`。
- 数据库的一个表对应一个领域类，以 `entity`、`domain` 或者 `meta` 作为包名都可以。
- 数据访问层命名形如 `xxxDao`，这里应该封装所有与数据库层相关的东西，如表名、各个列等。



- Service 类是封装业务逻辑的类，其中的方法要和此业务逻辑相关。比如 UserService 就是和 User 相关的业务方法。
- 当一个东西具有缓存和实际值的时候，务必保证存储和获取的接口只有一个。比如获取用户的 Level，应该只在 UserLevelService 中暴露一个接口提供。
- 当一个方法的主逻辑代码超过 30 行，务必考虑封装方法或者类。

当然，对于这些规范，可以使用 CheckStyle、PMD、FindBugs 等工具进行保证，使用相关的 Maven 插件即可。

- CheckStyle: 检查 Java 源文件是否与代码规范相符，主要包括以下几项。
  - Javadoc 注释。
  - 命名规范。
  - 多余没用的 Imports。
  - Size 度量，如过长的方法。
  - 缺少必要的空格 Whitespace。
  - 重复代码。

默认规范过于严格，可以参考这个文件做自定义规则: <https://github.com/superhj1987/awesome-libs/blob/master/doc/checkstyle-checks.xml>。

- PMD: 检查 Java 源文件中的潜在问题，主要包括以下几项。
  - 空 try/catch/finally/switch 语句块。
  - 未使用的局部变量、参数和 private 方法。
  - 空 if/while 语句。
  - 过于复杂的表达式，如不必要的 if 语句等。
  - 复杂类。

默认规范比较严格，可以参考这个文件做自定义规则: <https://github.com/superhj1987/awesome-libs/blob/master/doc/pmdrule.xml>。

- FindBugs: 基于 Bug Pattern 概念，查找 Java 字节码 (.class 文件) 中的潜在 Bug。主要检查字节码中的 Bug Pattern，如 NullPointerException、没有合理关闭资源、字符串相同判断错 (==, 而不是 equals) 等。大多数提示有用，值得改。可以借鉴

此文件，忽略一些不必要的规则：<https://github.com/superhj1987/awesome-libs/blob/master/doc/findbugs-exclude.xml>。

虽然以上工具已经可以告警不符合规范的代码，但仍然需要通过团队成员之间的代码审校才能够进一步保证开发规范的实践。

此外，这里还有一些开发建议。

- 务必撰写文档，尤其对于一些逻辑复杂的项目或者模块，需要包括以下文档：项目 / 模块的介绍文档；QuickStart，按照文档说明应该能在 1 小时内完成代码构建和简单使用；详细说明文档，比如主要逻辑流程、接口定义、参数含义、设计等。
- 在代码无法阐述设计意图的时候，注释是必需的。但注释不宜过多，过多意味着代码的可读性有问题。
- 防御性编程：处理异常、不要相信外部输入、不要相信外部依赖。
- 简洁与抽象：最好的代码抽象是对现实概念的映射。即只要有相关知识，每个类的作用都能在第一时间令人明白。但千万不要过度抽象，过度的抽象反而会降低代码质量。
- 代码的可重用：代码的可重用是优秀工程师的一个本能，但切记不能为了复用而复用。
- 程序需要状态，对象不需要状态。程序的状态应该由数据库、缓存、任务队列这些外部容器来统一容纳，处理时仅仅在对象的方法中以局部变量的形式循环。

# 第 3 章

## 开发框架

说到 Java 开发，开发框架是逃不开的话题。得益于 Java 生态圈中各种框架层出不穷，Java 的生命力才如此旺盛。在无数次被声称“Java 将亡”后 Java 依然活跃在开发领域，各种 Java 框架功不可没。

在 Java 刚开始出现的时候，写一个简单的类、配上 main 方法就能完成一个可运行的程序。然而随着应用规模越来越大以及参与人数增多，以往一个类能够完成的功能，现在需要许多类来完成。这时候如果没有一个开发规范或者说是约定，那么随着功能的增多，会越来越难以维护。此外，由于开发人员水平参差不齐，需要一些公共组件来保证一定的代码质量。开发框架就是一种基于约定、屏蔽了底层细节、规范了开发流程的技术。

在一个可用的 Java Web 应用中，一般由以下几种框架构成。

- **IoC 框架**：依赖注入 / 控制反转，即将依赖从代码层面转移到了容器配置层面。其是 Java 中使用得最普遍的框架。
- **ORM 框架**：对象关系映射，即将数据库的表映射到 Java 中对象的一种数据库操作框架。
- **Log 框架**：日志框架。即记录应用运行、异常日志的框架。
- **Web 框架**：一般指的是 Model-View-Controller 的分层 Web 开发框架，将业务代码做了逻辑分层，各司其职，能够做到灵活的配置和扩展。

Java Web 应用层次结构如图 3-1 所示。

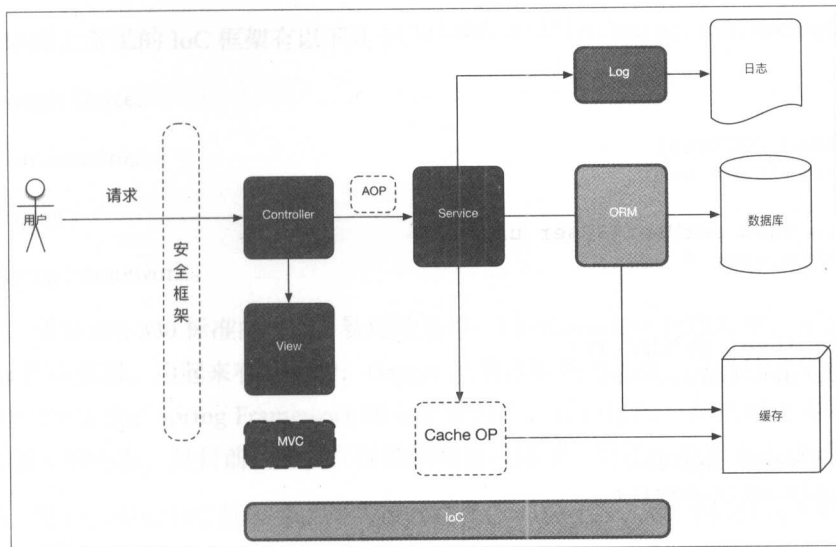


图 3-1

如图 3-1 所示，除了上述的 4 种框架，还有安全框架、AOP、缓存操作框架也是关键的开发框架。

这里需要说明的一点是，虽然使用开发框架有很多好处，但任何事情都不会是绝对的，都是一种权衡利弊后做出的选择。就 MVC 框架来说，当提供的业务接口需要特别高的并发量的时候，可以考虑不使用框架，直接使用 Servlet 处理请求并直接输出流数据（不走视图解析引擎和 JSON 等序列化）。毕竟请求经过框架的一层层调用，性能肯定会受到影响。其他的框架（诸如 ORM 框架）也与此类似。

### 3.1 依赖注入

IoC，控制翻转（Inversion of Control），又叫依赖注入（Dependency Inject）。即将代码里对象之间的依赖关系转移到容器中，这样就能够很灵活地通过面向接口的编程方式改变真正的实现类。

如下代码用于维护依赖关系，如果此时后面有了另外的 IUser 的实现类，那么要使用这个新的实现类则需要修改代码重新设置：

```

public interface IUser{
    void say();
}

public class AdminUser implements IUser{
    public void say(){

```

```

        System.out.println("I'm admin");
    }
}

public class IOCTest{
    private IUser user;

    public void setUser(IUser user){
        this.user = user;
    }

    public IUser getUser(){
        return this.user;
    }

    public void test(){
        this.user.say();
    }

    public static void main(String[] args){
        IOCTest test = new IOCTest();
        test.setUser(new AdminUser());

        test.test();
    }
}

```

而 IoC 的作用就是将这些依赖关系的维护从代码里拿出来，通过容器来维护。一个典型的例子，伪代码如下：

IOCContainer container = new IOCContainer("../"); // 可以通过一个上下文配置文件，也可以通过扫描注解

```

IOCTest iocTest = container.getInstance("iocTest");
iocTest.test();

```

一个典型的上下文配置文件如下：

```

<bean class="IOCTest" name="iocTest">
    <property name="user" ref="realUser"/>
</bean>
<bean class="AdminUser" name="realUser" />

```

这样在想要改变实现类时，只需要修改配置文件。

对于依赖注入，JSR-330 (Dependency Injection for Java) 做了一些规范。该规范主要是面向依赖注入使用者的，而对注入器实现、配置并未做详细要求。目前 Spring、Guice 已经开始兼容该规范。JSR-330 规范并未按 JSR 惯例发布规范文档，只发布了 API 源码。其指定了获取对象的一种方法，该方法与构造器、工厂以及服务定位器（例如 JNDI）这些传统方法相比可以获得更好的可重用性、可测试性以及可维护性。此方法的处理过程就是依赖注入。

目前市面上常见的 IoC 框架有以下几个。

- Google Guice。
- PicoContainer。
- Dagger。
- Spring Framework。

其中，兼容 JSR-330 标准的 Guice 易用性最好；PicoContainer 比较轻量，不过需要手工添加 Bean 类到容器，用起来有点复杂；Dagger 使用注解处理工具，其性能非常好，是一种很有前途的 DI 方案；Spring Framework 历史非常悠久，有自己的一套依赖注入体系，依赖于 Spring 强大的生态，是目前用得最广泛的依赖注入框架，且已经兼容 JSR-330 规范。

此外，基本所有的 IoC 框架都支持构造器注入、setter 注入以及字段注入 3 种方式。

### 3.1.1 JSR-330 依赖注入规范

如果要使用 JSR-330 提供的注解等功能，可引入依赖：

```
<dependency>
  <groupId>javax.inject</groupId>
  <artifactId>javax.inject</artifactId>
  <version>1</version>
</dependency>
```

主要提供了以下几个注解和类。

#### 1) @Inject

注解 @Inject 标识了可注入的构造器、方法或字段，可以用于静态或实例成员。一个可注入的成员可以被任何访问修饰符（private、package-private、protected、public）修饰。注入顺序为构造器、字段、方法。超类的字段、方法将优先于子类的字段、方法被注入。对于同一个类的字段是不区分注入顺序的，同一个类的方法亦同。如下：

```
public class IOCTest {
    private IUser user;
    @Inject
    public void setUser(IUser user) {
        this.user= user;
    }
}
```

## 2) @Qualifier

@Qualifier 是一个元注解，用来构建自定义限定符，任何人都可以定义新的限定器注解。一个限定器注解

- 是被 @Qualifier、@Retention (RUNTIME) 标注的，通常也被 @Documented 标注。
- 可以拥有属性。
- 可能是公共 API 的一部分，就像依赖类型，而不像类型实现，不作为公共 API 的一部分。
- 如果标注了 @Target 可能会有一些用法限制。本规范只指定了限定器注解可以被使用在字段和参数上，但一些注入器配置可能在其他一些地方（例如方法或类）上使用限定器注解。

## 3) @Named

@Named 就是使用上面介绍的 Qualifier 的一个限定器注解，可以指定依赖组件的名称：

```
public class IOCTest {  
  
    private IUser user;  
  
    @Inject  
    public void setUser(@Named("adminUser") IUser user) {  
        this.user= user;  
    }  
}
```

此外，@Named 还可以标注一个组件。如下：

```
@Named  
public class AdminUser implements IUser{  
    public void say(){  
        System.out.println("I'm admin").  
    }  
}
```

## 4) Provider&lt;T&gt;

接口 Provider 用于提供类型 T 的实例。Provider 一般情况是由注入器实现的。对于任何可注入的 T 而言，都可以注入 Provider<T>。与直接注入 T 相比，注入 Provider<T> 使得：

- 可以返回多个实例。
- 实例的返回可以延迟化或可选。
- 打破循环依赖。

- 可以在一个已知作用域的实例内查询一个更小作用域内的实例。

```
public class IOCTest {

    private IUser user;

    @Inject
    public void setUser(Provider<IUser> user) {
        this.user= user;
    }
}
```

### 5) @Scope

注解 @Scope 是一个元注解，用于标识作用域注解。一个作用域注解是被标识在包含一个可注入构造器的类上的，用于控制该类型的实例如何被注入器复用。默认情况下，如果没有标识作用域注解，注入器将为每一次注入都创建（通过注入类型的构造器）新实例，并不复用已有实例。如果多个线程能够访问一个作用域内的实例，该实例的实现应该是线程安全的。作用域实现由注入器完成。

### 6) @Singleton

@Singleton 是基于 Scope 注解实现的一个作用域注解，表示注入器只实例化一次的类型。该注解不能被继承。如下：

```
@Singleton
public class AdminUser implements IUser{
    public void say(){
        System.out.println("I'm admin").
    }
}
```

## 3.1.2 Guice

Guice 是 Google 开源的轻量级 IoC 框架，兼容 JSR-330 规范。其用 Module 来定义所有元件的实际类别。依赖定义部分可以使用 JSR-330 的注解。如下：

```
public class IOCTest {

    private IUser user;

    @Inject
    public void setUser(IUser user) {
        this.user= user;
    }
}
```



```

public class IOCTestModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(IUser.class).to(AdminUser.class);
        bind(IOCTest.class).to(IOCTest.class);
    }
}

public static void main(String[] args) {
    Injector injector = Guice.createInjector(new IOCTestModule());
    IOCTest test = injector.getInstance(IOCTest.class);
    ...
}

```

### 3.1.3 PicoContainer

PicoContainer 是一个“微核心”（micro-kernel）的容器，它利用了 Inversion of Control 模式和 Template Method 模式，提供面向组件的开发、运行环境。PicoContainer 是“极小”的容器，只提供了最基本的特性。其最重要的特性是实例化任意对象。这些通过它的 API 完成，这些 API 类似于 HashMap。向 PicoContainer 指定 java.lang.Class 对象，之后能够获得对象实例。如下：

```

MutablePicoContainer pico = new DefaultPicoContainer();

pico.addComponent(AdminUser.class); // 通过 class 注册
pico.addComponent(new AdminUser()) // 通过 type 注册
pico.addComponent(IOCTest.class);

IUser user = (IUser) pico.getComponent(AdminUser.class);
IOCTest test = (IOCTest)pico.getComponent(IOCTest.class);

```

不过目前 PicoContainer 的发展几乎已经停滞。笔者仅仅在 IntelliJ 的插件开发中见过对它的使用。

### 3.1.4 Dagger

Dagger 是 Google 开源的一个框架（早先的版本是由 Square 创建的，现版本由 Google 维护），支持 Android 和 Java，现在已经更新到 2.0 版本。它使用生成代码实现完整依赖注入的框架（在编译期），极大地减少了使用者的编码负担，且其相对于其他大部分 IoC 框架来说没有使用反射，性能有一定的提升。相对于同出自 Google 的 Guice 来说，其更加轻量级，但没有 Guice 中一些相对高级的功能，如 AOP 等。

Dagger 的依赖注入有自己的一些注解配置，如下：

```
public class IOCTest {

    private IUser user;

    @Inject
    public void setUser(IUser user) {
        this.user= user;
    }
}

@Module
public class TestModule {
    @Provides IUser provideUser() {
        return new AdminUser();
    }
}

@Component(modules = TestModule.class)
public interface TestComponent {
    void inject(IOCTest test);
}
```

使用如下：

```
IOCTest test = new IOCTest();
TestComponent component = DaggerActivityComponent.builder().
activityModule(new TestModule()).build();
component.inject(test);
test.test();
...
```

### 3.1.5 Spring Framework

Spring 的 IoC 应该是 Java 开发中最常用的功能之一。其 XML 配置的一个例子如下：

```
<!--applicationContext.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:p="http://www.springframework.org/schema/p"
        xmlns:c="http://www.springframework.org/schema/c"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean class="IOCTest" id="iocTest">
        <property name="user" ref="realUser"/>
    </bean>
    <bean class="AdminUser" id="realUser" />
</beans>
```

使用代码：

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationConte
xt("applicationContext.xml");
IOCTest iocTest = context.getBean(IOCTest.class);
iocTes.test();
```

此外，从 Spring 2.0 开始引入了对注解的支持，并且后来逐步兼容了 JSR-330 规范。这里简单对比以下 Spring 自身的依赖注入注解与 JSR-330 的，如表 3-1 所示。

表 3-1

Spring	JSR-330	备注
@Autowired	@Inject	@Inject 注解没有 required 属性
@Component	@Named	JSR-330 规范并没有提供复合的模型，只有一种方式来识别组件
@Scope("singleton")	@Singleton	JSR-330 默认的作用域类似 Spring 的 prototype，而 Spring 默认是单例的。如果要使用非单例的作用域，开发者应该使用 Spring 的 @Scope 注解。java.inject 也提供了一个 @Scope 注解，然而这个注解仅仅在创建自定义的作用域时才能使用
@Qualifier	@Qualifier/@Named	javax.inject.Qualifier 仅仅是一个元注解，用来构建自定义限定符。而 String 的 @Qualifier 等限定符可以通过 javax.inject.Named 来实现

除上述的注解外，Spring 还有注入 @Value、@Required 等注解，这在 JSR-330 中都没有对应的注解。

需要补充的一点是，Spring 的 IoC 目前早已支持 JSR-250（common annotation）中提供的注解：Resource、PostConstruct、PreDestroy。表 3-2 中与 Spring 自带的注解做一下对比。

表 3-2

Spring	JSR-250	备注
@Autowired	@Resource	@Resource 是先根据 Bean 的名称去匹配 Bean，获取不到的话再根据类型去匹配；而 @Autowired 则是根据类型匹配，通过名称则需要 Spring 的 @Qualifier 注解配合
@PostConstruct	init-method	Spring 中的 XML 配置中的 init-method 可以有同样的作用，即在 Bean 构造完后做一些初始化动作；@PostConstruct 具有更高优先级，同时存在时会先执行
@PreDesroy	destroy-method	Spring 中的 XML 配置中的 destroy-method 可以有同样的作用，即在 Bean 销毁前做一些收尾工作；@PreDesroy 注解具有更高优先级，同时存在时会先执行

这里需要说明的是，Spring 对 JSR-250 的支持的实现是在 `org.springframework.context.annotation.CommonAnnotationBeanPostProcessor` 类中。

### 3.1.6 循环依赖问题

IoC 中一个常见的问题就是循环依赖，如下：

```
class TestA {  
    @Inject TestB b;  
}  
  
class TestB {  
    @Inject TestC c;  
}  
  
class TestC {  
    @Inject TestA a;  
}
```

以上几个 IoC 框架，对于循环的处理：

- 通过替换为 `Provider<T>`，并且在构造器或者方法中调用 `Provider` 的 `get` 方法来打破循环依赖。
- 如果不依赖 `Provider`，对于构造器中的循环依赖是无法解决的，会抛出异常。
- 对于方法或字段注入的情况，将其依赖的一边放置到单例作用域中（可以缓存），使得循环依赖能够被注入器解析。

## 3.2 对象关系映射

ORM（Object Relational Mapping），对象关系映射，是一种为了解决面向对象与关系型数据库不匹配而出现的技術，使开发者能够用面向对象的方式使用关系型数据库。

目前最常用的 ORM 框架主要是 MyBatis（前身是 iBATIS）和 Hibernate。两者对比如下。

- MyBatis 非常简单易学，Hibernate 相对较复杂，门槛较高。
- 相比 Hibernate，MyBatis 灵活性更好。
- 在系统数据处理量巨大，性能要求极为苛刻的情况下，MyBatis 能够高度定制 SQL，因此会有更好的可控性和表现。

- MyBatis 需要手写 SQL 语句,也可以生成一部分, Hibernate 则基本上可以自动生成,偶尔会写一些 HQL。同样的需求, MyBatis 的工作量比 Hibernate 要大很多。类似地,如果涉及数据库字段的修改, Hibernate 要修改的地方很少,而 MyBatis 要对那些 SQL Mapping 的地方进行一一修改。
- 以数据库字段一一对应映射得到的 PO 与 Hibernate 这种对象化映射得到的 PO 是截然不同的,本质区别在于这种 PO 是扁平化的,不像 Hibernate 映射的 PO 是可以表达立体的对象继承、聚合等关系的,这将会直接影响到你的整个软件系统的设计思路。

Hibernate 现在主要用在传统企业应用的开发上,互联网领域中由于流量大、并发高的缘故主要以 MyBatis 为主。笔者也推荐使用 MyBatis 作为 ORM 框架。

一般的 ORM 包括以下几个部分。

- 一个规定 Mapping Metadata 的工具,即数据库中的表、列与对象以及对象属性的映射。
- 一个对持久类对象进行 CRUD 操作的 API。
- 一个语言或 API 用来规定与类和类属性相关的查询。
- 一种技术可以让 ORM 的实现同事务对象一起进行缓存、延迟加载等操作。

### 3.2.1 表元数据的映射

MyBatis 支持 XML 配置:

```
<!--mybatis-config.xml-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="${driver}"/>
        <property name="url" value="${url}"/>
        <property name="username" value="${username}"/>
        <property name="password" value="${password}"/>
      </dataSource>
    </environment>
  </environments>
  <mapppers>
    <mapper resource="TestUserMapper.xml"/>
  </mapppers>
```

```

</configuration>

<!--TestUserMapper.xml-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="me.rowkey.pje.mybatis.TestUserMapper">
  <select id="selectUser" resultType="TestUser">
    select * from test_user where id = #{id}
  </select>
</mapper>

```

其中, `environment` 元素体中包含了事务管理和连接池的配置, 能够根据不同的环境使用不同的数据库配置。在这里的 `dataSource` 被设置为 `POOLED` 时使用了 MyBatis 自己提供的数据库连接池。`mappers` 元素则是包含一组 Mapper 映射器 (这些 Mapper 的 XML 文件包含了 SQL 代码和映射定义信息)。

MyBatis 也支持注解映射 Mapper:

```

public interface TestUserMapper {

    @Select("SELECT * FROM test_user WHERE id = #{id}")
    TestUser selectUser(int id);

}

```

需要注意的是, 命名空间现在是必需的, 并且 MyBatis 对所有的命名配置元素的解析如果是全限定名则直接用; 而如果是简单的名称, 全局唯一的话没有问题, 如果有重复类则会报错。

此外, MyBatis 提供了 `mybatis-spring` 这个类库用于集成 Spring 和 MyBatis, 配置如下:

```

<!-- dataSource 数据源 -->
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
  <property name="username" value="${mysql.username}"></property>
  <property name="password" value="${mysql.pwd}"></property>
  <property name="maxTotal" value="${mysql.max}"></property>
  <property name="minIdle" value="${mysql.minIdle}"></property>
  <property name="maxIdle" value="${mysql.maxIdle}"></property>
  ...
</bean>

<!-- 配置 sqlSessionFactory 工厂 -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.
SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="configLocation" value="classpath:sqlMapConfig.xml"></
property>

```

```

</bean>

<bean id="sqlSessionTemplate" class="org.mybatis.spring.
SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory" />
</bean>

<bean id="userDAO" class="me.rowkey.pje.mybatis.dao.UserDao">
    <property name="sqlSessionTemplate" ref="sqlSessionTemplate"></
property>
</bean>

```

### 3.2.2 CRUD 以及属性的查询

MyBatis 的核心是 `SqlSessionFactory`，要使用它最基本的操作 API，首先获取 `SqlSessionFactory`，然后再获取相应的 `Mapper`。

```

String resource = "classpath:mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().
build(inputStream);

```

```

SqlSession session = sqlSessionFactory.openSession();
TestUserMapper mapper = session.getMapper(TestUserMapper.class);
TestUser testUser = mapper.selectUser(1);
...

```

此外，在 Spring 中可以使用 MyBatis 提供的 `org.mybatis.spring.SqlSessionTemplate` 进行操作：

```

SqlSessionTemplate sqlTemplate = new SqlSessionTemplate(sqlSessionFacto
ry);
TestUser testUser = (TestUser)this.sqlSessionTemplate.selectOne("me.
rowkey.pje.mybatis.TestMapper.selecUser", uid);

```

需要注意的是，`SqlSession` 是非线程安全的，而 `SqlSessionTemplate` 则是线程安全的。

MyBatis 对应于 CRUD 有以下几个映射语句。

- insert 映射插入语句。
- update 映射更新语句。
- delete 映射删除语句。
- select 映射查询语句。

依赖于上述的 4 种操作，可以完成各种 CRUD 以及对属性之类的查询。MyBatis 会自动把数据库查询结果注入返回对象中。

需要注意的是，上述的 `select * from test_user where id = #{id}`，其中的 `#{id}` 告诉 MyBatis 创建预处理语句属性并以它为背景设置安全的值，对应于 `PreparedStatement` 中的 `?`。此外，在 MyBatis 中还存在另一个符号 `$`，如下：

```
<select id="selectUserByStatus" resultType="TestUser">
    select * from test_user where status = #{status} order by ${columnName}
</select>
```

`$` 在这里的作用只是做字符串替换，不会修改或转义字符串。因此，能使用 `#` 的地方就不要使用 `$`，除非是像 `order by` 这种不是参数的地方。

此外，MyBatis 对于 `insert`、`update`、`delete` 以及 `select` 都提供了很多选项，用来支持注入缓存、自动映射、列名和属性名的转换等优化功能。

### 3.2.3 查询缓存的使用

MyBatis 支持一级缓存和二级缓存：

- Mybatis 默认开启一级缓存，其是 `SqlSession` 级别的，`Session` 结束缓存即清空。
- MyBatis 的二级缓存是 `Mapper` 级别的，需要手动开启。

```
<!--TestMapper.xml-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="me.rowkey.pje.mybatis.TestMapper">
    <cache/>
    <select id="selectUser" resultType="TestUser" useCache="false"
        flushCache="true">
        select * from test_user where id = #{id}
    </select>
</mapper>
```

上述配置中的 `useCache` 设置为 `true` 即开启二级缓存，对应 `Mapper` 中的所有 `select` 语句的结果集将会被缓存。也可以针对某个 `select` 设置 `useCache` 为 `false` 来关闭二级缓存，设置 `flushCache` 为 `true` 使得数据直接 flush 到数据库中防止出现脏读数据。此外，二级缓存也可以配置成使用自定义的缓存实现：

```
<cache type="me.rowkey.pje.mybatis.cache.MyCustomCache"/>
```



### 3.2.4 结果的映射

MyBatis 默认会自动映射查询结果，根据 SQL 返回的列名并在 Java 类中查找相同名字的属性（忽略大小写），还可以全局配置将数据库列的 LowUnderscore（a\_column）命名转换为 LowCamel 命名（aColumn）：

```
<configuration>
  <settings>
    <setting name="mapUnderscoreToCamelCase" value="true" />
  </settings>
</configuration>
```

除此之外，MyBatis 还支持自定义映射：

```
public class UserDto{
    private long uid;
    private String userName;

    //getter and setters
    ....
}

<mapper namespace="me.rowkey.pje.mybatis">
  <resultMap type="UserDto" id="userResultMap">
    <result property="userName" column="uname"/>
  </resultMap>

  <!-- 根据 uid 查询用户开放信息 -->
  <select id="getOpenUserInfo" parameterType="String"
resultType="UserDto" resultMap="userResultMap"
    useCache="true" flushCache="true">
    <![CDATA[
      select uid,uname from test_user
      WHERE uid = #{id}
    ]]>
  </select>
</mapper>
```

上面使用的 resultMap 即做了自定义的映射工作，uid 会自动映射，userName 会使用自定义映射。

### 3.2.5 规范 SQL 书写的语句构建器

虽然 MyBatis 已经做了很多封装，可以大大简化 SQL 编写工作，但是很多时候在代码中拼接 SQL 是无法避免的。如果用字符串自己进行拼接，那么各种 + 号、括号、引号、格式化问题会带来很多的麻烦，一不小心就会出错。针对这种状况，MyBatis 提供了 SQL 语句构建类 org.apache.ibatis.jdbc.SQL，可以大大简化动态 SQL 编写问题：

```
new SQL() {{
    SELECT("user.id, user.user_name");
    SELECT("user.sign, user.gender");
    FROM("user_account user");
    INNER_JOIN("user_base_info uinfo on user.uid = uinfo.uid");
    WHERE("user.user_name like ?");
    OR();
    WHERE("uinfo.nick_name like ?");
    ORDER_BY("user.create_time");
}}.toString();
```

等同于

```
"SELECT user.id, user.user_name, "
"user.sign, user.gender " +
"FROM user " +
"INNER JOIN user_base_info uinfo on user.uid = uinfo.uid" +
"WHERE (user.user_name like ?) " +
"OR (uinfo.nick_name like ?) " +
"ORDER BY user.create_time";
```

### 3.2.6 使用提示

- 当几个 SQL 语句都包含同样的部分 SQL 逻辑时，可以使用 `<include refid="id" />` 来进行复用。如下：

```
<sql id="getTestByStatus_fragment">
    from test_meta where status = #{status}
</sql>

<select id='getTestList' parameterType='map' resultMap='Test'>
    select id
    <include refid="getTestByStatus_fragment"/>
    order by
    create_time desc
</select>
```

- 动态 SQL 中的空字符串判断。

在动态 SQL 中判断是否是空字符串时，MyBatis 的内建机制不太好用，建议使用以下方式：

```
<if test="param != null and param != ''">
```

- 多 resultMap 复用。

一个应用中由于有不同功能经常会有多个 mapper.xml，如果一个文件需要另一个文件中的 resultMap 的定义，可以直接引用，而不需要再重新定义一遍。如下：

```

<!--A.mapper.xml-->
<mapper namespace="me.rowkey.pje.mybatis">
    ...
    <resultMap id="userResultMap" type="me.rowkey.pje.mybatis.
    UserDto"></resultMap>
</mapper>

<!--B.mapper.xml-->
<mapper namespace="me.rowkey.pje.test">
    ...
    <select id="bSql" parameterType="map" resultMap="me.rowkey.pje.
    mybatis.userResultMap>
    ...
    </select>
</mapper>

```

### 3.3 日志

日志在应用开发中是一个非常关键的部分。有经验的工程师能够凭借以往的经验判断出哪里该打印日志、该以何种级别打印日志，这样就能够在线上发生问题的时候快速定位并解决问题，极大地降低了应用的运维成本。

使用控制台输出其实也算日志的一种，在容器中会打印到容器的日志文件中。但是，控制台输出过于简单，缺乏日志中的级别控制、异步、缓冲等特性，因此在开发中要杜绝使用控制台输出作为日志（System.out.println）。而 Java 中已经有很多成熟的日志框架供大家使用，分别如下。

- JDK Logging。
- Apache Log4j。
- Apache Log4j2。
- Logback。

此外，还有两个用于实现日志统一的框架 Apache 的 Commons Logging、SLF4J。与上述框架的不同之处在于，这两个框架只是一个门面，并没有日志框架的具体实现，可以认为是日志接口框架。

对于这些日志框架来说，一般会解决日志中的以下问题。

- **日志的级别：**定义日志级别来区分不同级别日志的输出路径、形式等，帮助我们适应从开发调试到部署上线等不同阶段对日志输出粒度的不同需求。

- 日志的输出目的地：包括控制台、文件、GUI 组件，甚至是套接口服务器、UNIX Syslog 守护进程等。
- 日志的输出格式：日志的输出格式（JSON、XML）。
- 日志的输出优化：缓存、异步等。

这里需要说明的是，目前有几个框架提供了占位符的日志输出方式，然而其最终是用 `indexOf` 去循环查找并对信息进行拼接的，这样会消耗 CPU 资源。建议使用正确估算大小的 `StringBuilder` 拼装输出信息，除非是实在无法确定日志是否输出才用占位符。

### 3.3.1 JDK Logging

JDK Logging 就是 JDK 自带的日志操作类，在 `java.util.logging` 包下面，通常被简称为 JUL。

#### 配置

JDK Logging 配置文件默认位于 `$JAVA_HOME/jre/lib/logging.properties` 中，可以使用系统属性 `java.util.logging.config.file` 指定相应的配置文件对默认的配置文件的覆盖：

```
handlers= java.util.logging.FileHandler,java.util.logging.ConsoleHandler
.level= INFO #rootLogger 的日志级别

## 以下是 FileHandler 的配置
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter =java.util.logging.XMLFormatter #
配置相应的日志 Formatter。

## 以下是 ConsoleHandler 的配置
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter =java.util.logging.
SimpleFormatter # 配置相应的日志 Formatter。

# 针对具体的某个 Logger 的日志级别配置
me.rowkey.pje.log.level = SEVERE

# 设置此 Logger 不会继承上一级 Logger 的配置
me.rokey.pje.log.logger.useParentHandlers = false
```

这里需要说明的是，Logger 默认是继承配置的，如 `me.rowkey.pje.log` 的 Logger 会继承 `me.rowkey.pje` 的 Logger 配置，可以对 Logger 配置 `Handler` 和 `useParentHandlers`（默认是 `true`）属性，其中 `useParentHandler` 表示是否继承父 Logger 的配置。

JDK Logging 的日志级别比较多, 从高到低为: OFF( $2^{31}-1$ ) → SEVERE(1000) → WARNING(900) → INFO(800) → CONFIG(700) → FINE(500) → FINER(400) → FINEST(300) → ALL( $-2^{31}$ )。

## 使用

JDK Logging 的使用非常简单:

```
public class LoggerTest {
    private static Logger logger = Logger.getLogger(this.getClass().getName());

    public static void main(String argv[]) {
        logger.info("logger info");
    }
}
```

## 性能优化

JDK Logging 是一个比较简单的日志框架, 并没有提供异步、缓冲等优化手段, 也不建议大家使用此框架。

### 3.3.2 Log4j

Log4j 应该是目前 Java 开发中用得最广泛的日志框架。

## 配置

Log4j 支持 XML、Properties 配置, 通常使用 Properties:

```
root_log_dir=${catalina.base}/logs/app/

# 设置 rootLogger 的日志级别以及 appender
log4j.rootLogger=INFO,default

# 设置 Spring Web 的日志级别
log4j.logger.org.springframework.web = ERROR

# 设置 default appender 为控制台输出
log4j.appender.default=org.apache.log4j.ConsoleAppender
log4j.appender.default.layout=org.apache.log4j.PatternLayout
log4j.appender.default.layout.ConversionPattern=[%-d{HH:mm:ss} %-3r %-5p %l] >> %m (%t)%n

# 设置新的 Logger, 在程序中使用 Logger.get("myLogger") 即可
log4j.logger.myLogger=INFO,A2
```

```
# 设置另一个 appender 为按照日期轮转的文件输出
log4j.appender.A2=org.apache.log4j.DailyRollingFileAppender
log4j.appender.A2.File=${root_log_dir}log.txt
log4j.appender.A2.Append=true
log4j.appender.A2.DatePattern= yyyyMMdd'.txt'
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=[%-d{HH:mm:ss} %-3r %-5p %l]
>> %m (%t)%n
```

如果 Log4j 文件不直接存储于 classpath 下的话, 可以使用 PropertyConfigurator 来进行配置:

```
PropertyConfigurator.configure("...");
```

Log4j 的日志级别相对于 JDK Logging 来说, 简化了一些: DEBUG < INFO < WARN < ERROR < FATAL。

这里的 Logger 默认会继承父 Logger 的配置 (rootLogger 是所有 Logger 的父 Logger), 如上面 myLogger 的输出会同时出现在控制台和文件中出现。如果不想这样, 那么只需要进行如下设置:

```
log4j.additivity.myLogger=false 即可
```

## 使用

程序中对于 Log4j 的使用也非常简单:

```
import org.apache.log4j.Logger;
```

```
...
```

```
class Test{
    privat static final Logger LOGGER = Logger.getLogger(this.getClass());

    public void test(){
        LOGGER.info("...")
    }
}
...
```

这里需要注意的是, 虽然 Log4j 可以根据配置文件中日志级别的不同做不同的输出, 但由于字符串创建或者拼接也是耗资源的, 因此下面的用法是不合理的。

```
LOGGER.debug("...");
```

合理的做法应该是首先判断当前的日志级别是什么, 然后做相应的输出, 如下:

```
if(LOGGER.isDebugEnabled()){
    LOGGER.debug("...");
}
```

当然，如果是必须输出的日志可以不做此判断，比如 catch 异常打印错误日志的地方。

## 性能优化

Log4j 为了应对某一时间内大量的日志信息进入 appender 的问题，提供了缓冲来进一步优化性能。

```
log4j.appender.A3.BufferedIO=true
#Buffer 单位为字节，默认是 8KB，I/O BLOCK 大小默认也是 8KB
log4j.appender.A3.BufferSize=8192
```

以上表示当日志内容达到 8KB 时，才会将日志输出到日志输出目的地。

除了缓冲外，Log4j 还提供了 AsyncAppender 来做异步日志，但是 AsyncAppender 只能通过 XML 配置使用：

```
<appender name="A2"
  class="org.apache.log4j.DailyRollingFileAppender">
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%m%n" />
  </layout>
  <param name="DatePattern" value="'.'yyyy-MM-dd-HH" />
  <param name="File" value="app.log" />
  <param name="BufferedIO" value="true" />
  <!-- 8K 为一个写单元 -->
  <param name="BufferSize" value="8192" />
</appender>

<appender name="async" class="org.apache.log4j.AsyncAppender">
  <appender-ref ref="A2"/>
</appender>
```

### 3.3.3 Log4j2

2015 年 8 月，官方正式宣布 Log4j 1.x 系列生命终结，推荐用户升级到 Log4j2，并号称在修正了 Logback 固有的架构问题的同时，改进了许多 Logback 所具有的功能。Log4j2 与 Log4j 很不同，并不兼容，并且 Log4j2 不仅仅提供了日志的实现，也提供了门面，目的是统一日志框架。其主要包含以下两部分。

- log4j-api：作为日志接口层，用于统一底层日志系统。
- log4j-core：作为上述日志接口的实现，是一个实际的日志框架。

## 配置

Log4j2 的配置方式只支持 XML、JSON 以及 YAML，不再支持 Properties 文件，其配置文件的加载顺序如下。

- log4j2-test.json/log4j2-test.jsn。
- log4j2-test.xml。
- log4j2.json/log4j2.jsn 文件。
- log4j2.xml。

如果想要自定义配置文件位置，则需要设置系统属性 log4j.configurationFile:

```
System.setProperty("log4j.configurationFile", "...");
```

或者

```
-Dlog4j.configurationFile="xx"
```

配置文件示例:

```
<!--log4j2.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN" monitorInterval="30">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} -
      %msg%n"/>
    </Console>
    <File name="File" fileName="app.log" bufferedIO="true"
    immediateFlush="true">
      <PatternLayout>
        <pattern>%d %p %C{1.} [%t] %m%n</pattern>
      </PatternLayout>
    </File>
    <RollingFile name="RollingFile" fileName="logs/app.log"
      filePattern="log/${date:yyyy-MM}/app-%d{MM-dd-
      yyyy}-%i.log.gz">
      <PatternLayout pattern="%d{yyyy-MM-dd 'at' HH:mm:ss z} %-5level
      %class{36} %L %M - %msg%xEx%n"/>
      <SizeBasedTriggeringPolicy size="50MB"/>
      <!-- DefaultRolloverStrategy 属性如不设置，则默认为最多同一文件夹下 7 个文件，
      这里设置为 20 -->
      <DefaultRolloverStrategy max="20"/>
    </RollingFile>
  </Appenders>
  <Loggers>
    <logger name="myLogger" level="error" additivity="false">
      <AppenderRef ref="File" />
    </logger>
    <Root level="debug">
```



```

    <AppenderRef ref="Console"/>
  </Root>
</Loggers>
</Configuration>

```

上面的 `monitorInterval` 使配置变动能够被实时监测并更新，且能够在配置发生改变时不丢失任何日志事件；`additivity` 和 `Log4j` 一样也是为了让 `Logger` 不继承父 `Logger` 的配置；`Configuration` 中的 `status` 用于设置 `Log4j2` 自身内部的信息输出，当设置为 `trace` 时，会看到 `Log4j2` 内部各种详细输出。

`Log4j2` 在日志级别方面也有一些改动：`TRACE < DEBUG < INFO < WARN < ERROR < FATAL`，并且能够很简单地自定义自己的日志级别。

```

<CustomLevels>
  <CustomLevel name="NOTICE" intLevel="450" />
  <CustomLevel name="VERBOSE" intLevel="550" />
</CustomLevels>

```

上面的 `intLevel` 值是为了与默认提供的标准级别进行对照的。

## 使用

使用方式也很简单：

```
private static final Logger LOGGER = LogManager.getLogger(this.
getClass());
```

```
LOGGER.debug("log4j debug message");
```

需要注意的是，其中的 `Logger` 是 `log4j-api` 中定义的接口，而 `Log4j` 中的 `Logger` 则是类。

相比之前需要先判断日志级别，再输出日志，`Log4j2` 提供了占位符功能：

```
logger.debug("error: {} ", e.getMessage());
```

## 性能优化

在性能方面，`Log4j2` 引入了基于 `LMAX` 的 `Disruptor` 的无锁异步日志来实现进一步提升异步日志的性能：

```

<AsyncLogger name="asyncTestLogger" level="trace" includeLocation="true">
  <AppenderRef ref="Console"/>
</AsyncLogger>

```

需要注意的是，由于默认日志位置信息并没有被传给异步 `Logger` 的 I/O 线程，因此这里的 `includeLocation` 必须要设置为 `true`。

和 `Log4j` 一样，`Log4j2` 也提供了缓冲配置来优化日志输出性能。

```

<Appenders>
  <File name="File" fileName="app.log" bufferedIO="true"
  immediateFlush="true">
    <PatternLayout>
      <pattern>%d %p %C{1.} [%t] %m%n</pattern>
    </PatternLayout>
  </File>
</Appenders>

```

### 3.3.4 Logback

Logback 是由 Log4j 创始人设计的又一个开源日志组件，相对 Log4j 而言，在各个方面都有了很大改进。

Logback 当前分成 3 个模块。

- logback-core 是其他两个模块的基础模块。
- logback-classic 是 Log4j 的一个改良版本。logback-classic 完整实现 SLF4J API，使你  
可以很方便地更换成其他日志系统，如 Log4j 或 JDK14 Logging。
- logback-access 访问模块与 Servlet 容器集成提供通过 HTTP 来访问日志的功能。

### 配置

Logback 的配置文件如下：

```

<!--logback.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

  <property name="root_log_dir" value="${catalina.base}/logs/app/" />

  <appender name="FILE_APPENDER" class="ch.qos.logback.core.rolling.
  RollingFileAppender">
    <File>${root_log_dir}app.log</File>
    <Append>true</Append>
    <encoder>
      <pattern>%date [%level] [%thread] %logger{80} [%file : %line]
      %msg%n</pattern>
    </encoder>
    <rollingPolicy class="ch.qos.logback.core.rolling.
  TimeBasedRollingPolicy">
      <fileNamePattern>${root_log_dir}app.log.%d</fileNamePattern>
    </rollingPolicy>
  </appender>

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

```

```

        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
        </encoder>
    </appender>

    <logger name="myLogger" level="INFO" additivity="false">
        <appender-ref ref="FILE_APPENDER" />
    </logger>

    <root level="DEBUG">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>

```

Logback 的配置文件读取顺序（默认读取 classpath 下的文件）：logback.groovy → logback-test.xml → logback.xml。如果想要自定义配置文件路径，可以通过修改 logback.configurationFile 的系统属性：

```

System.setProperty("logback.configurationFile", "...");
或者
-Dlogback.configurationFile="xx"

```

Logback 的日志级别：TRACE < DEBUG < INFO < WARN < ERROR。如果 Logger 没有被分配级别，那么它将从被分配级别的最近的祖先那里继承级别。rootLogger 默认级别是 DEBUG。

Logback 中的 Logger 同样也是有继承机制的。配置文件中的 additivity 也是为了不继承 rootLogger 的配置，从而避免输出多份日志。

为了方便 Log4j 到 Logback 的迁移，官网提供了 log4j.properties 到 logback.xml 的转换工具：<https://logback.qos.ch/translator/>。

## 使用

Logback 由于天然与 SLF4J 集成，因此它的使用也就是 SLF4J 的使用：

```

import org.slf4j.LoggerFactory;

private static final Logger LOGGER=LoggerFactory.getLogger(this.
getClass());

LOGGER.info(" this is a test in {}", this.getClass().getName())

```

SLF4J 同样支持占位符。

## 性能优化

Logback 提供了 AsyncAppender 进行异步日志输出, 此异步 appender 实现上利用了队列做缓冲, 使得日志输出性能得到提高:

```
<appender name="FILE_APPENDER" class="ch.qos.logback.core.rolling.
RollingFileAppender">
    <File>${root_log_dir}app.log</File>
    <Append>true</Append>
    <encoder>
        <pattern>%date [%level] [%thread] %logger{80} [%file : %line]
%msg%n</pattern>
    </encoder>
    <rollingPolicy class="ch.qos.logback.core.rolling.
TimeBasedRollingPolicy">
        <fileNamePattern>${root_log_dir}app.log.%d</fileNamePattern>
    </rollingPolicy>
</appender>
<appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender">
    <discardingThreshold>0</discardingThreshold>

    <queueSize>512</queueSize>

    <appender-ref ref="FILE_APPENDER"/>
</appender>
```

这里需要特别注意以下两个参数的配置。

- queueSize: 队列的长度, 该值会影响性能, 需要合理配置。
- discardingThreshold: 日志丢弃的阈值, 即达到队列长度的多少会丢弃 TRACE、DEBUG、INFO 级别的日志, 默认是 80%, 设置为 0 则表示不丢弃日志。

### 3.3.5 统一日志 API 的门面框架

前面的 4 个框架是实际的日志框架。对于开发者而言, 每种日志都有不同的写法。如果我们以实际的日志框架来进行编写, 代码就限制死了, 之后很难再更换日志系统, 很难做到无缝切换。

Java 开发中经常提到面向接口编程, 所以我们应该按照一套统一的 API 来进行日志编程, 以实际的日志框架来实现这套 API, 这样的话, 即使更换日志框架, 也可以做到无缝切换。

这就是 Commons Logging 与 SLF4J 这种日志门面框架的初衷。

## Apache Commons Logging

Apache Commons Logging 经常被简称为 JCL，是 Apache 开源的日志门面框架。Spring 中使用的日志框架就是 JCL，使用起来非常简单：

```
import org.apache.commons.logging.LogFactory;

private static final Log LOGGER = LogFactory.getLog(this.getClass());

LOGGER.info("...");
```

使用 JCL 需要先引入 JCL 的依赖：

```
<dependency>
  <groupId>commons-logging</groupId>
  <artifactId>commons-logging</artifactId>
  <version>xx</version>
</dependency>
```

再来看一下如何让 JCL 使用其他日志实现框架。

1) 当没有其他日志 jar 包存在的时候，JCL 有自己的默认日志实现，默认的实现是对 JUL 的包装，即当没有其他任何日志包时，通过 JCL 调用完成的就 JUL 做日志操作。

2) 使用 Log4j 作为日志实现，那么只需要引入 Log4j 的 jar 包。

3) 使用 Log4j2 作为日志实现，那么除了 Log4j2 的 jar 包，还需要引入 Log4j2 与 Commons Logging 的集成包（使用 SPI 机制提供了自己的 LogFactory 实现）：

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-jcl</artifactId>
  <version>xx</version>
</dependency>
```

4) 使用 Logback 作为日志实现，那么由于 Logback 的调用是通过 SLF4J 的，因此需要引入 jcl-over-slf4j 包（直接覆盖了 JCL 的类），并同时引入 SLF4J 以及 Logback 的 jar 包。

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
  <version>xx</version>
</dependency>
```

## SLF4J

SLF4J（Simple Logging Facade for Java）为 Java 提供了简单日志 Facade，允许用户以自己的喜好，在工程中通过 SLF4J 接入不同的日志实现。与 JCL 不同的是，SLF4J 只提供接口，没有任何实现（可以认为 Logback 是默认的实现）。

SLF4J 的使用前提是引入 SLF4J 的 jar 包:

```
<!-- SLF4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>xx</version>
</dependency>
```

再看一下 SLF4J 如何和其他日志实现框架集成。

1) 使用 JUL 作为日志实现, 需要引入 slf4j-jdk14 包。

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-jdk14</artifactId>
  <version>xx</version>
</dependency>
```

2) 使用 Log4j 作为日志实现, 需要引入 slf4j-log4j12 和 Log4j 两个 jar 包。

```
<!-- slf4j-log4j -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>xx</version>
</dependency>

<!-- log4j -->
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>xx</version>
</dependency>
```

3) 使用 Log4j2 作为日志实现, 需要引入 log4j-slf4j-impl 依赖。

```
<!-- log4j2 -->
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>xx</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>xx</version>
</dependency>
<!-- log4j-slf4j-impl (用于 Log4j2 与 SLF4J 集成) -->
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j-impl</artifactId>
```

```
<version>xx</version>
</dependency>
```

4) 使用 Logback 作为日志实现，只需要引入 Logback 包。

### 3.3.6 统一日志框架的使用

上面说到了 4 种日志实现框架和 2 种日志门面框架。面对这么多的选择，即便是刚刚开始做一个应用，也会由于依赖的第三方库使用的日志框架五花八门而产生日志配置和使用上的烦恼。得益于 JCL 和 SLF4J，我们可以很容易地把日志统一为一种实现，从而进行集中配置和使用。这里就以用 Logback 统一日志实现为例。

1) 配置好 Logback 的依赖。

```
<!-- slf4j-api -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>xx</version>
</dependency>
<!-- logback -->
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-core</artifactId>
  <version>xx</version>
</dependency>
<!-- logback-classic (已含有对 slf4j 的集成包) -->
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>xx</version>
</dependency>
```

2) 切换 Log4j 到 SLF4J。

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>log4j-over-slf4j</artifactId>
  <version>xx</version>
</dependency>
```

3) 切换 JUL 到 SLF4J。

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jul-to-slf4j</artifactId>
  <version>xx</version>
</dependency>
```

#### 4) 切换 JCL 到 SLF4J。

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jul-over-slf4j</artifactId>
  <version>xx</version>
</dependency>
```

需要注意的是，做了以上配置后，务必要排除其他日志包的存在，如 Log4j。此外，在日常开发中经常由于各个依赖的库间接引入了其他日志库，造成日志框架的循环依赖。比如同时引入了 log4j-over-slf4j 和 slf4j-log4j12 的情况，当使用 SLF4J 调用日志操作时就会形成循环调用。

笔者目前比较推崇的是使用 SLF4J 统一所有框架接口，然后都转换到 Logback 的底层实现。但这里需要说明的是 Logback 的作者为了弥补 Log4j 的各种缺点而优化实现了 SLF4J 以及 Logback，但不知为何作者又推出了 Log4j2 以期取代 Log4j 和 Logback。所以，如果是一个新项目，那么直接跳过 Log4j 和 Logback 而选择 Log4j2 也是不错的选择，官网也提供了 Log4j 到 Log4j2 的迁移说明。

### 3.4 Web MVC

Web 开发最终肯定需要一个 Web 开发框架，而 MVC (Model View Controller) 是 Web 开发中最常用的框架。从 Struts1 到 Struts2，再到现在的 Spring MVC、Jersey 等，都是 MVC 模式的实现框架。MVC 将 Web 开发分为了模型、视图以及控制器 3 层，做到了职责分离，使应用的模块能够高内聚、低耦合。

- **模型：**代表业务数据和业务逻辑或者可以控制这些数据访问的模块。
- **视图：**对模型的展现。
- **控制器：**定义应用的各种行为。

目前，互联网领域主要以 Spring MVC 为主要的 Web 开发框架，因此本节主要讲述 Spring MVC 的使用。Spring 版本为 4.3.7.RELEASE。

#### 3.4.1 为什么是 Spring MVC

Spring MVC 是 Spring Web 的一个重要模块。其结构简单，强大不失灵活性，性能也很优秀。相比其他的框架，具有但不限于以下特点。

- 学习门槛低，易上手。



- 由于 Spring MVC 框架封装得比较好，因此使用 Spring MVC 很容易写出优秀的程序。
- Spring MVC 继承了 Spring 框架的灵活性，非常易于扩展。
- 框架的各个组件之间松耦合。
- 支持多种视图展现。
- 可以很方便地使用 Spring 生态下的组件。

### 3.4.2 Spring MVC 的请求处理流程

如图 3-2 所示是 Spring MVC 的几个关键组件。对于用户的 Web 请求的处理流程一般如下。

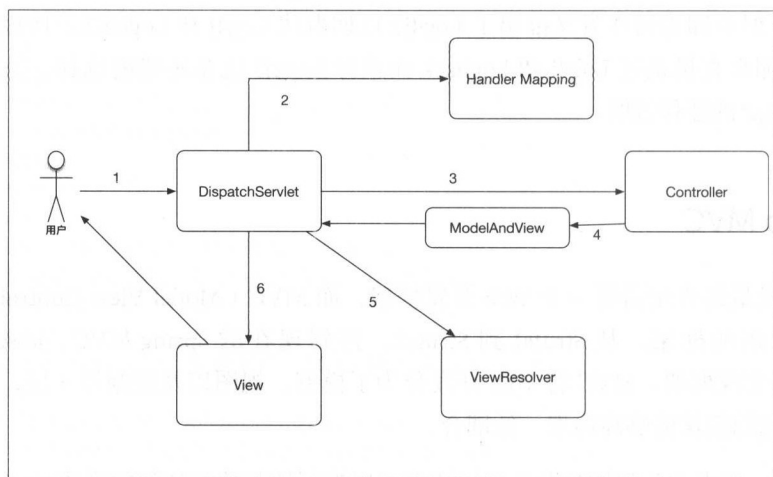


图 3-2

1) 用户发起请求到 DispatchServlet (在 web.xml 中配置，是 Spring MVC 的前置控制器)。

2) 从 Handler Mapping 中匹配此次请求信息的 Handler，匹配的条件包括：请求路径、请求方法、header 信息等。常用的几个 Handler Mapping 如下。

- SimpleUrlHandlerMapping：简单地映射一个 URL 到一个 Handler。
- RequestMappingHandlerMapping：扫描 RequestMapping 注解，根据相关配置，绑定 URL 到一个 Handler。

3) 获取对应的 Handler (Controller，控制器) 后，调用相应的方法。这里牵扯一个关键组件 HandlerAdapter，是 Controller 的适配器，Spring MVC 最终是通过 HandlerAdapter 来调用实际的 Controller 方法的。常用的有以下几个。

- SimpleControllerHandlerAdapter: 处理实现了 Controller 接口的 Controller。
- RequestMappingHandlerAdapter: 处理类型为 HandlerMethod 的 Handler, 这里使用 RequestMapping 注解的 Controller 的方法就是一种 HandlerMethod。

4) Handler 执行完毕, 返回相应的 ModelAndView。

5) 使用配置好的 ViewResolver 来解析返回结果。常用的几个 ViewResolver 如下。

- UriBasedViewResolver: 通过配置文件, 根据 URL 把一个视图名交给一个 View 来处理。
- InternalResourceViewResolver 类: 根据配置好的资源路径, 解析为 jsp 视图, 支持 jstl。
- FreeMarkerViewResolver: Freemarker 的视图解析器。

6) 生成视图返回给用户。常用的几个视图类如下。

- MappingJackson2JsonView: 使用 mappingJackson 输出 JSON 数据的视图, 数据来源于 ModelMap
- FreeMarkerView: 使用 FreeMarker 模板引擎的视图。
- JSTLView: 输出 jsp 页面, 可以使用 jsp 标准标签库。

此外, 还有 HandlerInterceptor 和 HandlerExceptionResolver 两个重要组件。

- HandlerInterceptor 是请求路径上的拦截器, 需要自己实现这个接口以拦截请求, 做一些对 Handler 的前置和后置处理工作。
- HandlerExceptionResolver 是异常处理器。其可以实现自己的处理器在全局层面拦截 Handler 抛出的 Exception, 再做进一步的处理。Spring MVC 自带了以下的几个处理器。
  - SimpleMappingExceptionResolver: 可以将不同的异常映射到不同的 jsp 页面。
  - ExceptionHandlerExceptionResolver: 解析使用了 @ExceptionHandler 注解的方法来处理异常。
  - ResponseStatusExceptionResolver: 处理 @ResponseStatus 注解的 Exception。
  - DefaultHandlerExceptionResolver: 默认的处理程序, 包括不支持 method、不支持 mediaType 等。

### 3.4.3 典型的配置方式

要使用 Spring MVC，首先需要配置 DispatcherServlet 负责分发所有请求：

```
<!--web.xml-->
<servlet>
    <servlet-name>mvc-dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath:mvc-dispatcher-servlet.xml
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup><!-- 是启动顺序，让这个 Servlet 随
Servlet 容器一起启动。-->
</servlet>
<servlet-mapping>
    <servlet-name>mvc-dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

上面的配置中，对应于 servlet-mapping 的 url-pattern 配置为 /，表示默认的 URL 映射，即当此次 request 匹配不到其他 Servlet 时，会默认进入此 Servlet，包括静态资源请求。此外，对应于 contextConfigLocation 参数的 mvc-dispatcher-servlet.xml 则是对 Spring MVC 的一些配置。如果想要使用 Spring 的注解配置，则可以如下这么配置：

```
<servlet>
    <servlet-name>mvc-dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
    <init-param>
        <param-name>contextClass</param-name>
        <param-value>org.springframework.web.context.support.AnnotationCon
figWebApplicationContext</param-value>
    </init-param>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            me.rowkey.config.SpringWebConfig
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

接着需要配置 MVC 的相关组件，如下：

```
<mvc:default-servlet-handler/>
```

```

<context:annotation-config/>

<mvc:annotation-driven>
    <mvc:message-converters>
        <bean class="org.springframework.http.converter.
ByteArrayHttpMessageConverter"/>
        <bean class="org.springframework.http.converter.
FormHttpMessageConverter"/>
        <bean class="org.springframework.http.converter.xml.
SourceHttpMessageConverter"/>
        <bean class="org.springframework.http.converter.json.
MappingJackson2HttpMessageConverter"/>
        <bean class="org.springframework.http.converter.
StringHttpMessageConverter">
            <constructor-arg value="UTF-8"/>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>

<bean class="org.springframework.web.servlet.view.
InternalResourceViewResolver">
    <property name="cache" value="false"/>
    <property name="prefix" value="/WEB-INF/jsp"/>
    <property name="suffix" value=".jsp"/>
    <property name="contentType" value="text/html;charset=UTF-8"/>
</bean>

<bean id="exceptionResolver"
    class="me.rowkey.exception.AppsExceptionResolver">
</bean>

<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.
CommonsMultipartResolver">
    <!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="1000000000"/>
</bean>

<bean
    class="org.springframework.context.support.
PropertyPlaceholderConfigurer">
    <property name="order" value="0"/>
    <property name="ignoreUnresolvablePlaceholders" value="true"/>
    <property name="locations">
        <list>
            <value>classpath:xx.properties</value>
        </list>
    </property>
</bean>

```

- 1) `mvc:default-servlet-handler` 是配置默认的 Servlet 作为静态资源的 Handler。
- 2) `context:annotation-config` 开启 Spring 的注解配置功能。
- 3) `mvc:annotation-driven` 是开启 MVC 的注解驱动，如创建了 `RequestMappingHandlerMapping` 和 `RequestMappingHandlerAdapter` 来处理注解 Handler。此外，上面配置了一些 `MessageConverter` 来处理各种使用了 `@ResponseBody` 标记返回数据的 Handler。
- 4) 配置了一个 `InternalResourceViewResolver` 处理返回数据。
- 5) 配置 `exceptionResolver` 来处理异常。
- 6) 配置 `CommonsMultipartResolver` 来处理文件上传。
- 7) 配置 `PropertyPlaceholderConfigurer` 来读取相关资源文件，从而在配置文件中可以使用占位符填充，在代码中可以使用 `@Value` 注解来引用 Properties 中的值。

此外，对应于 MVC 的 namespace，还有以下几个常用配置。

- `mvc:interceptors`: 配置拦截器。

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" />
    <bean class="me.rowkey.web.interceptor.MyInteceptor" />
  </mvc:interceptor>
</mvc:interceptors>
```

- `mvc:argument-resolvers`: 配置自己实现的参数解析器，可用于参数的名称转换，如当 API 传递的参数是 `underScore` 时，可以统一转换为 `lowCamel`。

这里需要注意的是，如果倾向于使用注解配置，那么对应于这些 XML 配置，Spring 都提供了相应的注解，可以参考官方文档。一个简单的注解配置如下：

```
@EnableWebMvc
@Configuration
@ComponentScan("me.rowkey.pje.web")
public class SpringWebConfig {

}

@ControllerAdvice
public class ExceptionCatcher {
    @ExceptionHandler
    @ResponseBody
    public String paramMissing(RuntimeException e) {
        ...
    }
}
```

上面的 `@ControllerAdvice` 是 Spring Web 3.2 后引入的注解，主要是为了统一对 Controller 添加 `@ExceptionHandler`、`@InitBinder`、`@ModelAttribute` 等注解，相比之前写一个 Base Controller 做好相关配置然后每一个 Controller 去继承的方式简化了很多。

### 3.4.4 无 XML 的配置方式

从 Servlet 3.0 开始，可以完全脱离 XML 对 Spring Web 项目进行配置，如下：

```
public class MyWebApplicationInitializer implements
    WebApplicationInitializer {

    @Override
    public void onStartUp(ServletContext appContext)
        throws ServletException {
        AnnotationConfigWebApplicationContext rootContext = new AnnotationConfigWebApplicationContext();
        rootContext.register(SpringWebConfig.class);

        appContext.addListener(new ContextLoaderListener(rootContext));

        ServletRegistration.Dynamic dispatcher = appContext.addServlet(
            "dispatcher", new DispatcherServlet(rootContext));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }
}
```

以上不需要再配置任何 XML 文件，即可使得 Spring MVC 项目得以运行。原理如下。

- Servlet 3.0 加入了一个特性：容器启动时会使用 Java 的 SPI（Service Provider Interface）机制读取 META-INF/services 下的 `javax.servlet.ServletContainerInitializer` 文件，并会对其中列出的每一个 `ServletContainerInitializer` 进行实例化，并调用 `onStartup` 方法。
- Spring Web 在自己的 `classpath:META-INF/services` 下的 `javax.servlet.ServletContainerInitializer` 文件中加入 `org.springframework.web.SpringServletContainerInitializer` 一行，于是此类被实例化并调用。
- `SpringServletContainerInitializer` 使用 `HandlesTypes` 声明自己处理实现了 `WebApplicationInitializer` 的类，于是上面我们新建的 `MyWebApplicationInitializer` 会被实例化并调用 `onStartup` 方法。

### 3.4.5 对 MVC 应用做单元测试

Spring MVC 支持对 Controller 的单元测试:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
    "classpath:mvc-dispatcher-servlet.xml",
})
@WebAppConfiguration
public class ControllerJUnitBase {

    @Resource
    private RequestMappingHandlerMapping handlerMapping;

    @Resource
    private RequestMappingHandlerAdapter handlerAdapter;

    /**
     * 执行 request 对象请求的 action
     *
     * @param request
     * @param response
     * @return
     * @throws Exception
     */
    public ModelAndView excuteAction(HttpServletRequest request,
        HttpServletResponse response) throws Exception
    {
        HandlerExecutionChain chain = handlerMapping.getHandler(request);

        final ModelAndView model = handlerAdapter.handle(request,
            response, chain.getHandler());
        return model;
    }

    @Test
    public void test throws Exception(){
        MockHttpServletRequest request = new MockHttpServletRequest();
        request.setRequestURI("/api/user/login");
        request.addParameter("mobile", "180xxxx3360");
        request.setMethod("POST");

        MockHttpServletResponse response = new MockHttpServletResponse();
        final ModelAndView mav = this.excuteAction(request, response);
        Assert.assertEquals("user_login", mav.getViewName());
    }
}
```

### 3.4.6 验证 Web 请求的参数

Web 开发中对前端传入的参数进行验证是一个关键的环节, 对于每一个 Controller 都单独做验证是一种方式, 但是更好的方式则是用一套框架将这个验证流程统一起来。

在 Spring Web 开发中, 常用的验证方式主要有以下两种。

- 支持 Spring 框架定义的 Validator 接口定义的校验。
- 支持 JSR-303 Bean Validation 定义的校验规范。

#### Spring Validator

这种方式是 Spring 框架自带的。

首先需要实现 `org.springframework.validation.Validator` 接口:

```
public class User{
    private String name;

    ...
}

public class UserValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return clazz.equals(User.class);
    }

    @Override
    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmpty(errors, "name", "user.name.
        required", "用户名不能为空");

        User user = (User)target;
        int length = user.getName().length();
        if(length > 10){
            errors.rejectValue("name", "user.name.too_long",
                "用户名不能超过 {20} 个字符");
        }
    }
}
```

其次, 需要设置 Validator 并触发校验。在 Controller 里增加方法并以 `@InitBinder` 注解, 在对应的 Controller method 中触发:

```
@InitBinder
protected void initBinder(WebDataBinder binder){
```



```

        binder.setValidator(new UserValidator());
    }

    @RequestMapping (method = RequestMethod.POST)
    public String reg(@Validated User user, BindingResult result){
        // 校验没有通过
        if(result.hasErrors()){
            return "user";
        }

        if(user != null){
            userService.saveUser(user);
        }

        return "user";
    }

```

如此，从页面提交的 User 对象可以通过我们实现的 UserValidator 类来校验，校验的结果信息存入 BindingResult 对象中。

## JSR-303 Bean Validation

在 Spring 3.1 中增加了对 JSR-303 Bean Validation 规范的支持，不仅可以对 Spring 的 MVC 进行校验，也可以对 Hibernate 的存储对象进行校验，是一个通用的校验框架。

这里必须要引入 hibernate-validator，并开启 MVC 注解支持（<mvc:annotation-driven />），它是 JSR-303 规范的具体实现。

此外，需要对要校验的 meta 类的属性做注解 Constraint 限制。JSR-303 定义的 Constraint 如下。

- @Null：验证对象是否为空。
- @NotNull：验证对象是否为非空。
- @AssertTrue：验证 Boolean 对象是否为 true。
- @AssertFalse：验证 Boolean 对象是否为 false。
- @Min：验证 Number 和 String 对象是否大于或等于指定的值。
- @Max：验证 Number 和 String 对象是否小于或等于指定的值。
- @DecimalMin：验证 Number 和 String 对象是否大于或等于指定的值，需要注意小数的精度问题。

- @DecimalMax: 验证 Number 和 String 对象是否小于或等于指定的值, 需要注意小数的精度问题。
- @Size: 验证对象 (Array、Collection、Map、String) 长度是否在给定的范围内。
- @Digits: 验证 Number 和 String 的构成是否合法。
- @Past: 验证 Date 和 Calendar 对象是否在当前时间之前。
- @Future: 验证 Date 和 Calendar 对象是否在当前时间之后。
- @Pattern: 验证 String 对象是否符合正则表达式的规则。

此外, hibernate-validator 也提供了一些注解支持, 如下。

- @NotEmpty: 验证对象不为 NULL 也不为 empty。
- @NotBlank: 验证对象不为 NULL 也不为 empty, 连续的空格也被认为是 empty。
- @Range: 验证对象在指定的范围内。

配置很简单, 只要对被校验的 meta 注解 Constraint 即可:

```
public class User{
    @NotNull
    private String name;

    ...
}
```

然后, 在 Controller 的对应方法中, 给对应的参数加 @Valid 注解:

```
public String doRegister(@Valid User user, BindingResult result){

    // 校验没有通过
    if(result.hasErrors()){
        return "user";
    }

    if(user != null){
        userService.saveUser(user);
    }

    return "user";
}
```

这样就可以完成针对输入数据 User 对象的校验了, 校验结果保存在 BindingResult 对象中。需要注意的一点是, BindingResult 参数如果放在验证参数的后面, 那么错误信息是会绑定到此 BindingResult 上的, 否则会抛出 MethodArgumentNotValidException 异常。

### 3.4.7 使用异步 Servlet

Servlet 3.0 引入了异步 Servlet，即 Connector 的线程只负责将请求派发到业务逻辑线程池，业务逻辑处理完成后再通过 Servlet 的异步上下文句柄将结果返回并响应，能够避免 Web Server 的连接池被长期占用而引起性能问题。Spring MVC 提供了对异步 Servlet 的支持。

1) 在 web.xml 启用异步支持。

```
<filter>
    <filter-name>Set Character Encoding</filter-name>
    <filter-class>org.springframework.web.filter.
CharacterEncodingFilter</filter-class>
    <async-supported>true</async-supported>
    ...
</filter>
<filter-mapping>
    <filter-name>Set Character Encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<servlet>
    <servlet-name>mvc-dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</
servlet-class>
    ...
    <async-supported>true</async-supported>
</servlet>
```

这里需要注意除了 Servlet 外，也要把所有经过的 filter 的 async-supported 都设置为 true。

2) 实现异步 Controller。

返回结果为 java.util.concurrent.Callable 即为异步 controller：

```
@Controller
@RequestMapping("/async")
public class CallableController {
    @RequestMapping("/test")
    @ResponseBody
    public Callable<String> callable() {

        return new Callable<String>() {
            @Override
            public String call() throws Exception {
                Thread.sleep(1000);
                return "Asyn Controller Result";
            }
        };
    }
}
```

### 3) 配置异步 Controller 使用的线程池和超时参数。

```
<task:executor id="myExecutor" pool-size="7-42" queue-capacity="11"/>

<mvc:annotation-driven>
    <mvc:async-support task-executor="myExecutor" default-timeout="2000"/>
</mvc:annotation-driven>
```

如此，处理业务的线程池即为 myExecutor，业务线程超时时间为 2 秒。

此外，如果想要针对具体的业务分别使用不同的线程池，那么可以通过返回 org.springframework.web.context.request.async.DeferredResult 进行：

```
private Executor executor = Executors.newFixedThreadPool(200); // 业务线程池

@RequestMapping("/deferred")
@ResponseBody
public DeferredResult<String> quotes() {
    DeferredResult<String> deferredResult = new
DeferredResult<String>(2000L); // 设置超时时间为 2 秒
    deferredResult.onCompletion(new Runnable() {
        @Override
        public void run() {
            System.out.println("Deferred Result done !!!");
        }
    });

    executor.execute(new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            deferredResult.setResult("Deferred Result");
        }
    });

    return deferredResult;
}
```

如此，请求将会被挂起，直到在其他线程中将 DeferredResult 放入数据才会响应，并且能够给 DeferredResult 设置回调方法，在数据返回后做相应的后续操作。

还需要注意的是，当使用异步 Servlet 时，Tomcat 等 Servlet 容器的线程池大小可以设置为 1：

```
<Connector port="8080" protocol="org.apache.coyote.http11.
Http11AprProtocol"
```

```

...
maxThreads="1"
minSpareThreads="1"
.../>

```

### 3.4.8 使用提示

- 如果遇到一个项目既提供了 JSON API 又提供了 Web 页面，那么单单配置一个 `ViewResolver` 是不行的。这时可以使用 Spring MVC 的视图协商器 `ContentNegotiatingViewResolver`。配置如下：

```

<bean id="contentNegotiatingViewResolver"
      class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="contentNegotiationManager">
        <bean class="org.springframework.web.accept.
ContentNegotiationManager">
            <constructor-arg>
                <list>
                    <bean class="org.springframework.web.accept.
PathExtensionContentNegotiationStrategy">
                        <constructor-arg>
                            <map>
                                <entry key="html" value="text/html;charset=UTF-8"/>
                                <entry key="json" value="application/
json;charset=UTF-8"/>
                                <entry key="xls" value="application/vnd.ms-excel"/>
                                <entry key="pdf" value="application/pdf"/>
                            </map>
                        </constructor-arg>
                    </bean>
                    <bean class="org.springframework.web.accept.
HeaderContentNegotiationStrategy"/>
                    <bean class="org.springframework.web.accept.FixedContentN
egotiationStrategy">
                        <constructor-arg value="application/json;charset=UTF-8">
                        </constructor-arg>
                    </bean>
                </list>
            </constructor-arg>
        </bean>
    </property>
    <property name="viewResolvers">
        <list>
            <bean
                class="org.springframework.web.servlet.view.
InternalResourceViewResolver">
                <property name="cache" value="false"/>
                <property name="prefix" value="/WEB-INF/jsp/">

```

```

        <property name="suffix" value=".jsp"/>
        <property name="contentType" value="text/
html; charset=UTF-8"/>
    </bean>
</list>
</property>
<property name="defaultViews">
    <list>
        <bean
            class="org.springframework.web.servlet.view.json.
MappingJackson2JsonView">
            <property name="objectMapper" ref="objectMapper"/>
            <property name="contentType" value="application/
json; charset=UTF-8"/>
            <property name="modelKeys">
                <set>
                    <value>data</value>
                    <value>status</value>
                    <value>desc</value>
                </set>
            </property>
        </bean>
    </list>
</property>
</bean>

```

上面 ContentNegotiationManager 中配置的 PathExtensionContentNegotiationStrategy 表示根据 path 的扩展名来匹配返回的数据格式，如 \*.json 就返回 JSON 数据格式。此外，还配置了 HeaderContentNegotiationStrategy，根据 header 里的 accept 信息匹配返回数据格式，最后配置一个 FixedContentNegotiationStrategy，作为以上都无效时的默认返回数据格式。接着，ContentNegotiatingViewResolver 又配置了一个解析为 jsp 页面的 ViewResolver 来处理返回格式为 text/html 的请求，如果此 ViewResolver 匹配不上，那么最后使用默认视图 defaultViews 来对 ModelMap 的值做 JSON 解析，上面的配置则是仅仅取 ModelMap 中对应于 data、status 以及 desc 的值作为 JSON 的视图字段。

- 如果使用视图解析器解析 Handler 的数据并返回响应视图，那么当 Controller 中的参数具有 HttpServletResponse 时，假如 Controller 没有返回值，则此次请求是不会走到视图解析器的。如下：

```

@RequestMapping(value = "test", method = RequestMethod.GET, headers
= "Accept=text/html")
public void testApi(ModelMap modelMap, HttpServletRequest request,
HttpServletResponse response, String id, int status){
    .....
}

```

Spring 中对于这种含有 `HttpServletResponse` 参数的 `Controller`，认为视图是由 `Handler` 自己生成的。如果仍然想要走视图解析器，则必须要返回一个值。

- API 请求返回 JSON、JSONP、XML 数据的时候，可以通过 `@ResponseBody` 来注解 `Controller` 防范，并配置好对应的 `MessageConvertor`。
- Spring 的 `Controller`、`Service`、`DAO` 等都是单实例的，因此 `Controller`、`Service`、`DAO` 等各层组件应该被设计为有行为无状态、有方法无属性，即使有属性，也只是对下一层组件的持有。而与之对比，项目中的 `Entity`、`Domain`、`DTO` 等各种实体，有状态无行为、有属性无方法，即使有方法，也只是 `getter` 和 `setter` 等，围着状态打转。
- `org.springframework.web.servlet` 路径下的 `DispatcherServlet.properties` 中配置了 Spring MVC 兜底使用的组件。即当项目的配置中缺少某一类组件的时候，Spring MVC 会使用此文件中的相应组件来顶替。

# 第 4 章

## Spring

在第 3 章的 IoC 以及 MVC 两节中已经提到了 Spring 的一部分功能。Spring 可以说是 Java 中用得最普遍的解决企业应用的一站式框架，基本上覆盖了 Java 开发中各种基础组件。可以说掌握了 Spring 框架的使用方法，就掌握了 Java 开发一半甚至更多的技能。

如图 4-1 所示来自 Spring 官方，可见 Spring 目前覆盖了很多方面的技术。Spring 本来是为了取代重量级的 EJB，结果现在自身变得越来越庞大，同时也导致复杂性显著提高。使用 Spring 的时候，应该做到全面了解，从而选择需要使用的，而不是大而全地都拿来用。而 Spring 对于插拔式的需求，支持得也比较好。

Spring 的常用组件，可见图 4-2。

- Spring 必需的组件，包括 Bean 容器、Context 上下文支持、Spring EL 以及相关支持工具类库。这些是 Spring 的基础组件，是其他所有组件依赖的基础。
- Spring AOP 是做日志统一管理、事务管理时需要用到的组件。
- Spring ORM 中提供了对 Hibernate 等 ORM 框架的整合组件。
- Spring JDBC 的 JdbcTemplate 是在做数据库操作时经常用到的组件。
- Spring TX 提供了数据库事务管理相关的组件。
- Spring Web 提供了 Web 开发时用到的诸如 Web 工具、视图解析器等组件。其通常与 Spring Web MVC 一起使用。



Spring Projects		
Reactor Core	Reactor Project	Spring AMQP
Spring Batch	Spring Boot	Spring Cloud CLI
Spring Cloud Cluster	Spring Cloud Commons	Spring Cloud Config
Spring Cloud Connectors	Spring Cloud Consul	Spring Cloud Contract
Spring Cloud Data Flow for Apache Mesos	Spring Cloud Data Flow for Apache YARN	Spring Cloud Data Flow for Cloud Foundry
Spring Cloud Data Flow for Kubernetes	Spring Cloud for Amazon Web Services	Spring Cloud Netflix
Spring Cloud Pipelines	Spring Cloud Sleuth	Spring Cloud Spinnaker
Spring Cloud Stream	Spring Cloud Stream App Starters	Spring Cloud Task
Spring Cloud Task App Starters	Spring Cloud Vault	Spring Cloud Vault
Spring Data Commons	Spring Data Envers	Spring Data for Apache Solr
Spring Data GemFire	Spring Data JDBC Extensions	Spring Data JPA
Spring Data LDAP	Spring Data MongoDB	Spring Data Neo4J
Spring Data Redis	Spring Data REST	Spring for Android
Spring for Apache Hadoop	Spring Framework	Spring HATEOAS
Spring Integration	Spring IO Platform	Spring LDAP
Spring Mobile	Spring REST Docs	Spring Roo
Spring Security	Spring Security Kerberos	Spring Security OAuth
Spring Session	Spring Shell	Spring Social
Spring Social Facebook	Spring Social Twitter	Spring Statemachine
Spring Test HtmlUnit	Spring Vault	Spring Web Flow
Spring Web Services	Spring XD	

图 4-1

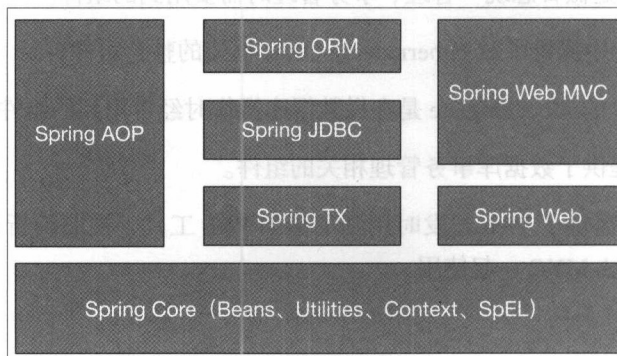


图 4-2

以上是 Spring 框架的核心组成，官方称之为 Spring Framework。除此之外，还有几个 Spring 项目也比较常用。

- **Spring Data:** 包括 Spring Data Redis、Spring Data MongoDB、Spring Data Solr 等，它们是对很多数据存储软件的操作封装。
- **Spring Boot:** 是 Spring 所有组件的集合，旨在简化 Spring 应用的配置和降低 Spring 使用的复杂度。
- **Spring Security:** 提供了权限管理、OAuth 开发等基础组件。

需要注意的是，本章讲述的 Spring 版本是 4.3.x 系列，是 Spring 4.x 系列的最终功能版本。而 Spring 最新的 5.x 版本带来了很多重大的更新，包括对 JDK 9、Servlet 4、HTTP 2、JUnit 5 以及完整的端到端响应式编程的支持。引入了 Router Functions、Spring WebFlux、HTTP/Reactive Streams 等组件来简化对非阻塞、事件驱动的网络应用程序的开发。

## 4.1 Spring 核心组件

Spring 框架的核心包括 IoC、AOP 以及辅助工具 SpringEL 等。

首先要明确一个概念，Spring 的 IoC 容器是 ApplicationContext。常用的 ApplicationContext 如下。

- **ClassPathXmlApplicationContext:** 从 ClassPath 路径中加载 XML 配置的上下文。
- **FileSystemXmlApplicationContext:** 从文件系统中加载 XML 配置的上下文。
- **XmlWebApplicationContext:** Web 开发中从 XML 中记载 Web 上下文，区别于上面两个之处在于，此上下文是基于 ServletContext 的。
- **AnnotationConfigWebApplicationContext:** 从注解类中加载 Web 上下文。

这些上下文中的实例在 Spring 中被叫作 Bean。一个 Bean 的生命周期管理如图 4-3 所示。

如图 4-3 所示为 Spring 中的几个关键类、接口以及对应的方法，箭头的方向表示执行顺序，描述了一个 Bean 从开始初始化到销毁会经历的方法调用。

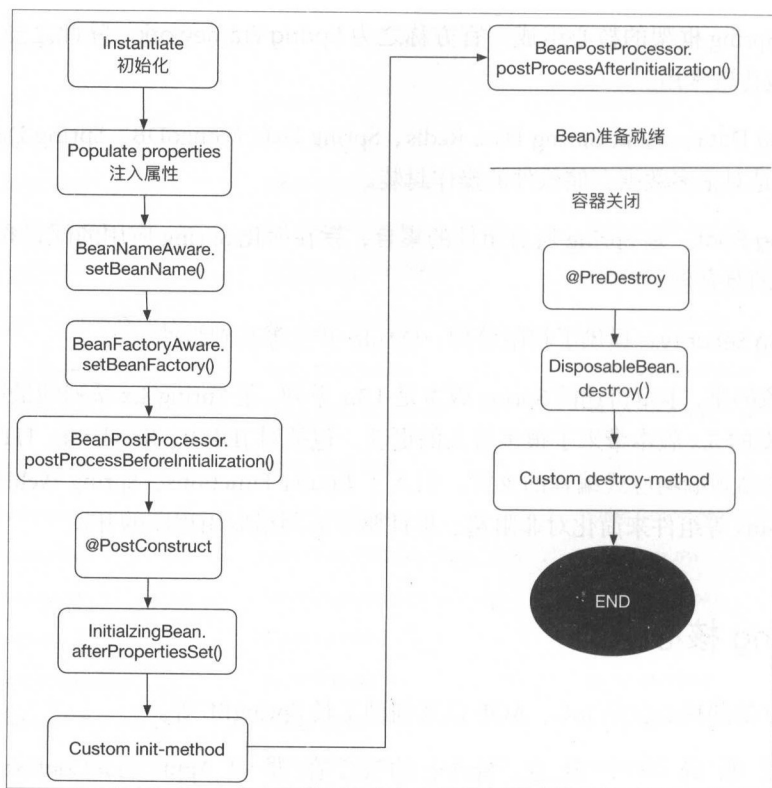


图 4-3

### 4.1.1 Spring 的双亲上下文机制

Spring 中的 `ApplicaitonContext` 是父子层次结构的，在存在多个上下文的时候，会有一个根上下文作为其他上下文的“父亲”：

```

<listener>
  <listener-class>org.springframework.web.context.
  ContextLoaderListener</listener-class>
  ...
</listener>

```

如上，如果在 `web.xml` 中使用 `Listener` 监听器来加载 Spring 的配置，Spring 会创建一个全局的 `WebApplicationContext` 上下文，其被称为根上下文，保存在 `ServletContext` 中，key 是 `WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE` 属性的值。可以使用工具类取出上下文 `WebApplicationContextUtils.getWebApplicationContext(ServletContext)` 或者 `WebApplicationContextUtils.getRequiredWebApplicationContext(ServletContext)`。

而在很多情况下, 我们还会配置一个或者多个 `DispatcherServlet`, 每个 `DispatcherServlet` 都有一个自己的 `WebApplicationContext` 上下文。这个上下文是私有的, 继承了根上下文中的所有东西。该上下文保存在 `ServletContext` 中, key 是 `"org.springframework.web.servlet.FrameworkServlet.CONTEXT" + Servlet 名称`。当一个 `Request` 对象产生时, 会把这个 `WebApplicationContext` 上下文保存在 `Request` 对象中, key 是 `DispatcherServlet.class.getName() + ".CONTEXT"`。可以使用工具类取出上下文 `RequestContextUtils.getWebApplicationContext(request)` 或者 `WebApplicationContextUtils.getWebApplicationContext(servletContext, attrname)`。

为了避免双亲上下文, 可以不使用 `Listener` 监听器加载 Spring 的配置, 直接改用 `DispatcherServlet` 加载 Spring 的配置。

### 4.1.2 Spring 中的事件机制

Spring 中提供了事件机制, 用于监听容器事件的发生, 在事件发生时做一些处理工作。

1) 容器事件监听器: 实现 `ApplicationListener` 接口的类, 可以监听容器的事件, 包括如下几种。

- `ContextStartedEvent`: 上下文启动事件。
- `ContextRefreshedEvent`: 上下文刷新完毕事件。
- `ContextStoppedEvent`: 上下文停止事件。
- `ContextClosedEvent`: 上下文关闭事件。

```
@Service
public class TestListener implements ApplicationListener<ContextStartedEvent> {
    @Override
    public void onApplicationEvent(ContextStartedEvent event) {
        ...
    }
}
```

```
<bean class="me.rowkey.pje.spring.TestListener" scope="prototype" />
```

2) 具有意识的 Bean: 实现了形如 `xxAware` 接口的 Bean 即为有意识的 Bean, 能够注入 / 获取 `xx` 代表的事物。如, 实现 `ApplicationContextAware` 接口的类, 会自动注入当前的 `ApplicationContext`。

```
public class ApplicationContextHolder implements ApplicationContextAware {
    private static ApplicationContext applicationContext;
```

```

    public static ApplicationContext getContext() {
        return ApplicationContextHolder.applicationContext;
    }

    public void setApplicationContext(ApplicationContext
applicationContext) throws BeansException {
        ApplicationContextHolder.applicationContext = applicationContext;
    }
}

<bean class="me.rowkey.pje.spring.ApplicationContextHolder"/>

```

此外，在 Bean 生命周期图中的 BeanNameAware 和 BeanFactoryAware 也是两个具有意识的 Bean。

- 实现了 BeanNameAware 的 Bean 能够感知到自己在 BeanFactory 中注册的名称。
- 实现了 BeanFactoryAware 的 Bean 能够感知到自己所属的 BeanFactory。

### 4.1.3 Bean 的初始化和销毁

如果想要在应用初始化的时候做一些初始化方法，可以使用以下几种初始化方式。

- 直接在 Bean 的构造方法里做初始化工作。
- 使用 @PostConstruct 注解，指明在 Bean 构造器方法执行后执行的方法。
- Bean 实现 InitializingBean 接口，在 afterPropertiesSet 中做初始化工作。
- 在 XML 中使用 init-method 指定 Bean 构造完成后调用的方法。

需要注意的是，上面的第 1 种方法和第 2 种方法并不保证在执行时依赖的其他 Bean 已经注入进来。因此如果想在某些 Bean 初始化完毕并注入进来之后再进行初始化工作，可以配合使用 @DependsOn 注解：

```

@Service
@DependsOn("configService")
public class AccountService implements Constants {

    @Resource
    private ConfigService configService;

    @PostConstruct
    public void init() throws IOException {
        ...
    }
}

```

也可以使用 `BeanFactoryPostProcessor` 和 `BeanPostProcessor` 来做一些更前置的初始化工作，典型的应用场景就是实现自己的注解。

- 要实现 `BeanFactoryPostProcessor` 接口，可以在 Spring 容器加载了 Bean 的定义之后，在 Bean 实例化之前执行，这样能够修改 Bean 的定义属性。如可以把 Bean 的 scope 从 `singleton` 改为 `prototype`，也可以把 `property` 的值修改掉。可以通过接口的参数获取相关 Bean 的定义信息。
- 实现 `BeanPostProcessor` 接口可以在 Spring 容器实例化 Bean 之后，在执行 Bean 的初始化方法前后，添加一些自己的处理逻辑。Spring 内置了几个 `BeanPostProcessor` 实现，分别如下。
  - `CommonAnnotationBeanPostProcessor`：支持 `@Resource` 注解的注入。
  - `RequiredAnnotationBeanPostProcessor`：支持 `@Required` 注解的注入。
  - `AutowiredAnnotationBeanPostProcessor`：支持 `@Autowired` 注解的注入。
  - `ApplicationContextAwareProcessor`：用来为 Bean 注入 `ApplicationContext` 等容器对象。

根据 Bean 的生命周期。以上提到的初始化方法的优先级为：`BeanFactoryPostProcessor` > `Constructor` > `BeanPostProcessor.postProcessBeforeInitialization` > `@PostConstruct` > `InitializingBean` > `init-method`。

此外，如果要在所有 Bean 都初始化完毕后做一次初始化工作，那么可以使用 4.1.2 节所介绍的 `ApplicationListener`，监听 `ContextRefreshedEvent`：

```
@Service
public class BootstrapService implements ApplicationListener<ContextRefreshedEvent> {
    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        ...// 初始化代码
    }
}
```

而要在销毁 Bean 之前做一些收尾工作，有以下 3 种方式。

- 使用 `@PreDestroy` 注解，指明在容器关闭后执行的方法。
- 实现 `DisposableBean` 接口，在 `destroy` 方法中做销毁工作。
- 在 XML 中使用 `destroy-method` 指定 Bean 销毁时调用的方法。

根据 Bean 的生命周期，可知优先级为：@PreDestroy > DisposableBean > destroy-method。

#### 4.1.4 Bean 的动态构造

当要创建的 Bean 不能直接通过构造方法、setter 方法、字段注入完成，还需要做一些初始化工作的时候，普通创建 Bean 的方式就力不从心了。Spring 提供了 3 种方式解决这个问题。

1) 定义 Bean 时，指明 factory-bean 和 factory-method。

```
public class TestFactory{
    public User getUser(){
        return new User();
    }
}

<bean id="testFactory" class="me.rowkey.pje.spring.TestFactory"/>
<bean id="user" class="me.rowkey.pje.common.meta.User"
    factory-bean="testFactory" factory-method="getUser">
</bean>
```

2) 定义 Bean 直接使用类的静态方法。

```
public class TestFactory{
    public static User getStaticUser(){
        return new User();
    }
}

<bean id="user" class="me.rowkey.pje.spring.TestFactory" factory-
method="getStaticUser"/>
```

3) 实现 FactoryBean 接口。

```
public interface FactoryBean<T> {
    T getObject() throws Exception;

    Class<?> getObjectType();

    boolean isSingleton();
}

public class TestFactory implements FactoryBean<User> {
    @Override
    public User getObject() throws Exception {
        return new User("test");
    }
}
```

```

@Override
public Class<?> getObjectType() {
    return User.class;
}

@Override
public boolean isSingleton() {
    return true;
}
}

<bean id="user" class="me.rowkey.pje.spring.TestFactory"/>

```

针对第3种方式的 FactoryBean, Spring 自带了几个实现。

- **PropertyPathFactoryBean**: 用来获取目标 Bean 的属性值（实际是其 getter 方法的返回值）。PropertyPathFactoryBean 返回值（基本数值类型或者对象）在最外层，则把返回值注册为容器中一个 Bean, Bean 名字为 id。

```

<bean id="propertyBean" class="org.springframework.beans.factory.
config.PropertyPathFactoryBean">
    <property name="targetBeanName" value="adminUser"></property>
    <property name="propertyPath" value="user.username"></property>
</bean>

```

<!-- 下面是简化形式 -->

```

<bean id="adminUser.user.username" class="org.springframework.beans.
factory.config.PropertyPathFactoryBean" />

```

- **FieldRetrievingFactoryBean**: 用来获取类的静态属性值或者对象的实例属性值。

```

<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE" class="org.
springframework.beans.factory.config.FieldRetrievingFactoryBean"/>

```

上面即获取类 java.sql.Connection 的静态属性值 TRANSACTION\_SERIALIZABLE。

- **MethodInvokingFactoryBean**: 用来调用类的静态方法或者调用 Bean 对象的实例方法。若有返回值则可注册为容器中的 Bean 或者作为依赖注入其他 Bean 中。

```

<bean id="testUser" class="org.springframework.beans.factory.
config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="testFactory"></property>
    <property name="targetMethod" value="getUser"></property>
    <property name="arguments">
        <list>
            .....
        </list>
    </property>
</bean>

```



同样地, 可以用 inner Bean 的方式将上面的 FactoryBean 构造的实例注入到其他 Bean 中:

```
<bean id="testUser" class="me.rowkey.pje.common.meta.User">
  <property name="username">
    <bean id="adminUser.user.username" class="org.springframework.
beans.factory.config.PropertyPathFactoryBean"/>
  </property>
  <property name="password">
    <bean id="adminUser.user.password" class="org.springframework.
beans.factory.config.PropertyPathFactoryBean"/>
  </property>
</bean>
```

当然, 现在 Spring 提供了 @Bean 注解, 完全可以在代码中做这种动态生成 Bean 的工作:

```
@Configuration
public class SpringCoreConfig{

    @Bean("testUser")
    public User TestUser(AdminUser adminUser) { // 参数可以直接引用 Bean

        User user = new User();
        user.setUsername(adminUser.getUser().getUsername());

        ...

        return user;
    }
}
```

#### 4.1.5 注入集合、枚举、类的静态字段

1) 对于在 XML 中已经存在的 Bean 注入 list、Map 等集合这种需求, 可以如下这样配置。

```
<!-- Bean 的属性是集合 -->
<bean id="collectionBean" class="me.rowkey.pje.spring.CollectionBean">
  <property name="list">
    <list>
      <value>list1</value>
      <value>list2</value>
      <value>list3</value>
    </list>
  </property>
  <property name="map">
    <map>
      <entry key="key1" value="value1"></entry>
      <entry key="key2" value="value2"></entry>
      <entry key="key3" value="value3"></entry>
    </map>
  </property>
</bean>
```

## 2) 注入枚举、静态值。

对于枚举和静态值的注入,使用 FieldRetrievingFactoryBean 是一种办法。但其实当注入的字段值属于此字段对应类的静态字段时,只需要注明静态字段名即可。如下:

```
public enum Status{
    ENABLE,DISABLE
}

public class ParentUser{
    public static final ParentUser INSTANCE = new ParentUser();
}

public class User{
    private Status status;

    private ParentUser parentUser;

    private void setStatus(Status status){
        this.status = status;
    }

    private Status getStatus(){
        return this.status;
    }

    private void setParentUser(ParentUser parentUser){
        this.parentUser = parentUser;
    }

    private ParentUser getParentUser(){
        return this.parentUser;
    }
}

<bean id="testUser" class="me.rowkey.pje.common.meta.User">
    <property name="status" value="ENABLE"/>
    <property name="parentUser" value="INSTANCE"/>
</bean>
```

### 4.1.6 面向方面编程——AOP

AOP (Aspect Oriented Programming) 是为了解决某些场景下代码重复问题的一种编程技术,允许程序模块化横向切割关注点或横向切割典型的责任划分。其能够封装多个类中不同单元的相同功能,把应用业务逻辑和系统服务分开,经常用在日志和事务管理上。

说到 AOP 不得不提的就是 AspectJ 这个 AOP 框架，它是一种编译期 AOP 框架（即在编译的时候对被代理的类进行增强）。它自己有一套 AOP 各个组件的概念定义。其中的几个关键的概念如下。

- **方面**：Aspect，横切多个类的某个功能描述。比如，一个日志模块可以被称作日志的 AOP 切面。
- **连接点**：JoinPoint，程序执行过程中的某个函数调用。其代表一个应用程序的某个位置，在这个位置我们可以插入一个 AOP 切面，它实际上是一个应用程序执行 Spring AOP 的位置。
- **通知**：Advice，是一个在方法执行前或执行后要做的动作，实际上是程序执行时要通过 Spring AOP 框架触发的代码段。Spring 切面可以应用以下 5 种类型的通知。
  - **before**：前置通知，在一个方法执行前被调用。
  - **after**：在方法执行之后调用的通知，无论方法执行是否成功。
  - **after-returning**：仅当方法成功完成后执行的通知。
  - **after-throwing**：在方法抛出异常退出时执行的通知。
  - **around**：在方法执行之前和之后调用的通知。
- **切入点**：PointCut，是映射到一个或一组连接点的指示符，通知将在这些位置执行。可以通过表达式或匹配的方式指明切入点。Spring AOP 支持的 AspectJ 切入点指示符如下。
  - **execution**：用于匹配方法执行的连接点。
  - **within**：用于匹配指定类型内的方法执行。
  - **this**：用于匹配当前 AOP 代理对象类型的执行方法。注意是 AOP 代理对象的类型匹配，这样就可能包括引入接口的类型匹配。
  - **target**：用于匹配当前目标对象类型的执行方法。注意是目标对象的类型匹配，这样就不包括引入接口的类型匹配。
  - **args**：用于匹配当前执行的方法传入的参数为指定类型的执行方法。
  - **@within**：用于匹配所有持有指定注解类型内的方法。
  - **@target**：用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解。
  - **@args**：用于匹配当前执行的方法传入的参数持有指定注解的执行方法。

- **@annotation**: 用于匹配当前执行方法持有指定注解的方法。
- **bean**: Spring AOP 扩展的, AspectJ 没有此指示符, 用于匹配特定名称的 Bean 对象的执行方法。
- **reference pointcut**: 表示引用其他命名切入点, 只有 @AspectJ 风格支持, Schema 风格不支持。

其中类型匹配的语法如下。

- **\***: 匹配任何数量字符。
- **..**: 匹配任何数量字符的重复, 如在类型模式中匹配任何数量子包, 而在方法参数模式中匹配任何数量参数。
- **+**: 匹配指定类型的子类型, 仅能作为后缀放在类型模式后边。

匹配类型表达式: 注解? 类的全限定名字; 匹配方法执行表达式: 注解? 修饰符? 返回值类型 类型声明? 方法名 (参数列表) 异常列表?

此外, 多个切入点表达式是可以组合的, AspectJ 使用且 (&&)、或 (||)、非 (!) 来组合切入点表达式, 为了方便 XML 配置, Spring AOP 提供了 and、or、not 来代替 &&、||、!。

需要注意的是, Spring AOP 是一种运行期的 AOP 框架, 对 AspectJ 的支持只是使用了其定义的各个组件的定义和注解, 底层实现和它并没有多少关系。

使用 Spring AOP 有如下 3 种方式。

#### 1) Spring AOP 接口。

这种方式基于 Spring 提供的 AOP 接口。

- 前置通知: MethodBeforeAdvice
- 后置通知: AfterReturningAdvice
- 环绕通知: MethodInterceptor
- 异常通知: ThrowsAdvice

针对这些接口, 进行实现并配置即可。这种方式使用起来比较复杂, 笔者也不推荐使用, 因此不再详细叙述。

## 2) 依赖于 AspectJ 的 XML 配置。

Spring 从 2.0 开始支持 AspectJ 的 XML 配置方式, 在 <aop:config> 标签下使用 <aop:pointcut>、<aop:advisor>、<aop:aspect> 完成配置, 大大简化了 AOP 的开发工作, 如下:

```
public class TestServiceAop{
    public void doAround(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("=====进入 around 环绕方法! ===== \n");

        // 调用目标方法之前执行的动作
        System.out.println(" 调用方法之前 : 执行! \n");

        // 调用方法的参数
        Object[] args = pjp.getArgs();
        // 调用的方法名
        String method = pjp.getSignature().getName();
        // 获取目标对象
        Object target = pjp.getTarget();
        // 执行完方法的返回值: 调用 proceed() 方法, 就会触发切入点方法执行
        Object result = pjp.proceed();

        System.out.println(" 输出: " + args[0] + ";" + method + ";" +
target + ";" + result + "\n");
        System.out.println(" 调用方法结束: 之后执行! \n");
    }
}

<bean id="testServiceAop" class="me.rowkey.pje.spring.aop.TestServiceAop"
/>

<aop:config>
    <aop:aspect ref="testServiceAop">

        <aop:pointcut id="point_cut" expression="execution(* me.rowkey.
pje.spring.aop.test.service.impl...*(..))" />

        <aop:before .../>

        <aop:afterReturning .../>

        <aop:afterThrowing .../>

        <aop:after .../>

        <aop:around method="around" pointcut-ref="point_cut"/>

        <!--<aop:around method="around" pointcut="execution(* me.rowkey.
pje.spring.service.impl...*(..))"/>-->
    </aop:aspect>
</aop:config>
```

## 3) 依赖于 AspectJ 注解的 AOP 配置。

Spring 现在也支持使用 AspectJ 的注解来做 AOP, 比起 XML 配置的方式, 这种方式更加简单:

```
@Component
@Aspect
public class UserRestOperationLogAop {

    @Pointcut("execution(public * me.rowkey.pje.spring.
service..*.add*(..)) " +
        "|| execution(public * me.rowkey.pje.spring.
service..*.delete*(..)) " +
        "|| execution(public * me.rowkey.pje.spring.
service..*.update*(..))" +
        "|| execution(public * me.rowkey.pje.spring.
service..*.create*(..))" +
        "|| execution(public * me.rowkey.pje.spring.
service..*.modify*(..))")
    public void pointCut() {
    }

    @Around(value = "pointCut()")
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        System.out.println("=====进入 around 环绕方法! ===== \n");

        // 调用目标方法之前执行的动作
        System.out.println("调用方法之前: 执行! \n");

        // 调用方法的参数
        Object[] args = pjp.getArgs();
        // 调用的方法名
        String method = pjp.getSignature().getName();
        // 获取目标对象
        Object target = pjp.getTarget();
        // 执行完方法的返回值: 调用 proceed() 方法, 就会触发切入点方法执行
        Object result = pjp.proceed();

        System.out.println("输出: " + args[0] + ";" + method + ";" +
target + ";" + result + "\n");
        System.out.println("调用方法结束: 之后执行! \n");
    }
}

<context:component-scan base-package="me.rowkey.pje.spring" />
<!-- 打开 AOP 注解 -->
<aop:aspectj-autoproxy />
```

Spring 中 AOP 的实现原理涉及两种方式, 分别是 JDK 动态代理和 CGLIB。

- 如果目标对象实现了接口, 默认情况下会采用 JDK 的动态代理实现 AOP。

- 如果目标对象实现了接口，可以强制使用 CGLIB 实现 AOP。
- 如果目标对象没有实现接口，必须采用 CGLIB 库，Spring 会自动在 JDK 动态代理和 CGLIB 之间转换。

也可以强制 AOP 使用 CGLIB，如下：

```
// XML 配置
<aop:aspectj-autoproxy proxy-target-class="true"/>

// 注解配置
@EnableAspectJAutoProxy(proxyTargetClass = true)
```

4.1.7 进阶 XML 的配置

XML 是 Spring 一开始就支持的配置方式，也是最主流的配置方式。从 Spring 2.0 开始，基于文件的配置就从 DTD 转换到了 XSD，其目的就是让大家能够更好地利用 XSD 做到配置的灵活性和便利性。这里首先要注意的是，在加载 XML 文件时，通常会有前缀 classpath: 或者 classpath\*: 这两个前缀的区别在于当 classpath 中存在同样路径的多个文件时，前者只会加载第一个找到的资源，而后者会加载所有找到的资源。路径的对照表如表 4-1 所示。

表 4-1

前缀	例子	说明
classpath:	classpath:applicationContext.xml	从 classpath 中加载
file:	file:/data/applicationContext.xml	作为 URL 从文件系统中加载
http:	http://host/applicationContext.xml	作为 URL 加载
( none )	/data/applicationContext.xml	根据上下文的不同而不同：FileSystemXmlApplicationContext 对应文件系统，GenericWebApplicationContext 对应 ServletContext 中的资源，ClassPathXmlApplicationContext 对应 classpath 下的资源

第 3 章 IoC 和 MVC 两节都有一些基本的 Spring XML 配置示例，这里主要讲解一些可能不是所有人都知道的知识点。首先看下面的配置示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean class="me.rowkey.test.TestBean" p:name="uname" p:refBean-
ref="refBean">
```

```

        <property name="type" value="1"/>
    </bean>
    <beans profile="dev">
        <bean id="bean2" class="me.rowkey.pje.spring.TestBean"
c:name="testName">
            <property name="refBean">
                <null/>
            </property>
        </bean>
    </beans>
</beans>

```

对上述示例配置的说明如下。

- `xsi:schemaLocation`: 为了指出 XSD 文件可以在本地 jar 包中找到, 无须去网络位置获取。
- `p:xx`: 对属性赋值的简化, 作用于属性注入。
- `c:xx`: 对构造注入的简化。
- 使用 `<null/>` 引用 NULL 值。
- 从 Spring 3.0 开始, `beans` 多了一个 `profile` 属性, 其可以根据不同的 `profile` 使用不同的 Bean, 可以用在本地、测试、生产环境使用不同的日志、数据库配置这种场景下。上面的 Bean2 表示在 `profile` 为 `dev` 时, 才会初始化这个 Bean。其中, `profile` 通过 `spring.profiles.active` 和 `spring.profiles.default` 这两个 Java 运行属性或者 `SPRING_PROFILES_DEFAULT` 和 `SPRING_PROFILES_ACTIVE` 环境变量来指定。此外, 对于 `ContextLoaderListener`, 在 `web.xml` 中可以配置其默认 `profile`:

```

<context-param>
    <param-name>spring.profiles.default</param-name>
    <param-value>local</param-value>
</context-param>

```

对于 `DispatchServlet`, 在 `web.xml` 中进行配置:

```

<servlet>
    <init-param>
        <param-name>spring.profiles.default</param-name>
        <param-value>local</param-value>
    </init-param>
</servlet>

```

除了 `beans` 外, Spring 还提供了很多额外的 namespace 以简化配置。

- `context`: 此 namespace 下的配置主要是对上下文的配置简化。



- **annotation-config**: 开启注解装配。隐式地向 Spring 容器注册 `AutowiredAnnotationBeanPostProcessor`、`RequiredAnnotationBeanPostProcessor`、`CommonAnnotationBeanPostProcessor` 以及 `PersistenceAnnotationBeanPostProcessor` 这 4 个 `BeanPostProcessor`。
- **component-scan**: 指定注解扫描的包路径。此配置包含了 `annotation-config` 的功能，因此如果配置了 `component-scan`，那么 `annotation-config` 是可以省略的。
- **property-holder**: 根据指定的 `location` 扫描 `properties` 文件以及系统属性和环境变量，存储里面的键值对，在 XML 配置里可以使用占位符 `${}` 引用。在代码中，可以使用注解 `@Value + 占位符引用`。
- **util**: 此 namespace 下主要提供了一些构建 Bean 的工具。
  - **constant**: 根据表达式创建指定静态字段作为 Bean，可替代 `FieldRetrievingFactoryBean` 的使用。
  - **property-path**: 获取指定 `path` 的属性作为 Bean，可替代 `PropertyPathFactoryBean` 的使用。
  - **properties**: 从指定位置加载 `properties` 文件，创建一个 `java.util.Properties` 实例 Bean（后面可以引用此 Bean 的属性），可替代 `PropertiesFactoryBean` 使用。
  - **list**: 创建一个列表 Bean，可替代 `ListFactoryBean` 使用。
  - **map**: 创建一个 map Bean，可替代 `MapFactoryBean` 使用。
  - **set**: 创建一个 set Bean，可替代 `SetFactoryBean` 使用。
- **MVC**
  - **annotation-driven**: 开启 MVC 的注解支持，创建了和 MVC、注解相关的一系列 Bean，如 `RequestMappingHandlerMapping`、`RequestMappingHandlerAdapter` 等。
    - ◆ **message-convertors**: `annotation-driven` 的子元素，用来配置 `HttpMessageConverter`。
    - ◆ **argument-resolvers**: `annotation-driven` 的子元素，用来配置项目用到的自定义 `HandlerMethodArgumentResolver`。
  - **interceptors**: 配置项目用到的拦截器。
  - **content-negotiation**: `view-resolvers` 的子元素，配置内容协商视图。

```

<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean class="org.springframework.web.servlet.view.
json.MappingJackson2JsonView"/>
        </mvc:default-views>
    </mvc:content-negotiation>
</mvc:view-resolvers>

```

### 4.1.8 无 XML 的配置方式

现在利用注解 + 代码配置变得越来越流行。基本上每一个 XML 配置都有相应的注解配置与之对应。常用的注解配置如下。

- **@Configuration**: 表示此类的用途是定义 Bean。
- **@Bean**: 注解到方法上, 创建、初始化一个 Bean, 对应于 <beans/> 配置。
- **@Profile**: 对应于 <beans profile='xxxx'> 指定某个 profile 激活的 Bean 等。
- **@ComponentScan**: 对应于配置, 扫描对应的包。
- **@EnableWebMVC**: 对应于配置, 开启 MVC 注解支持。配合 @Configuration 使用。被注解的对象实现 WebMvcConfigurer 接口 (一般继承 WebMvcConfigurerAdapter 即可), 可以自定义配置拦截器、视图协商器、MessageConverters 等。
- **@Import**: @Import 允许将其他 JavaConfig 形式的配置类引入到当前的 @Configuration 标注的配置类当中, 对应于原来 XML 中的 <import/>, 也可以通过 @ImportResource 将 XML 形式定义的配置引入当前 JavaConfig 形式的配置类中。
- **@PropertySource**: 配合 @Configuration 使用, 用来加载 .properties 内容到 Environment, 比如 @PropertySource("classpath:config.properties"), 需要容器中配置一个 PropertySourcesPlaceholderConfigurer。

一个配置示例如下:

```

@Configuration
@ImportResource({
    "classpath:applicationContext-core.xml"
})

@Import(SpringExtraConfig.class)
@ComponentScan(basePackages = {
    "me.rowkey.pje.spring.mvc"
}, excludeFilters = {

```



## 4.2 Spring 数据操作框架

Spring 提供了对常用数据软件 / 服务操作的封装组件, 包括关系型数据库、NoSQL 数据库、Redis、Elasticsearch、Cassandra 等。本章主要讲述其中最常用的 JDBC、Redis 以及 MongoDB。

### 4.2.1 Spring JDBC

3.2 节介绍了 ORM 框架, 其实 Spring 也提供了对 ORM 的支持, 包括对 Hibernate、JDO 以及 JPO 的支持。相关组件的层次结构如图 4-4 所示。

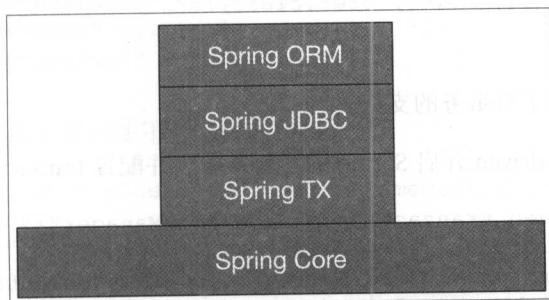


图 4-4

比较常用的是 Spring ORM 下面一层的 Spring JDBC 和 Spring TX。

1) Spring JDBC 提供了对 JDBC 操作的封装, 也是 Java 开发中经常用到的数据库操作工具, 经常需要在需要灵活组装 SQL 的场景下使用, 其核心类是 `JdbcTemplate`, 提供了很多数据库 CRUD 操作。

注入一个数据库连接池即可构造 `JdbcTemplate`:

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
  <property name="url" value="${mysql.url}"/>
  <property name="username" value="${mysql.user}"/>
  <property name="password" value="${mysql.pwd}"/>
  <property name="maxTotal" value="64"/>
  <property name="maxWaitMillis" value="3000"/>
  <property name="maxIdle" value="32"/>
  <property name="minIdle" value="10"/>
</bean>

<bean id="jdbcTemplate"
  class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

注入 JdbcTemplate 到 DAO 中, 可以进行数据库操作:

```
@Repository
public class UserDao {
    @Resource
    private JdbcTemplate jdbcTemplate;

    public User getById(long id) {
        return jdbcTemplate.queryForObject("select * from test_user where
id = ?", new Object[]{id}, User.class);
    }

    public List<User> getByName(String name) {
        return jdbcTemplate.queryForList("select * from test_user where
name= ?", new Object[]{name}, User.class);
    }
}
```

## 2) Spring TX 提供了对事务的支持。

使用 tx:annotation-driven 开启 Spring 的注解事务, 并配置 transaction-manager:

```
<tx:annotation-driven transaction-manager="txManager"/>

<bean id="txManager" class="org.springframework.jdbc.datasource.
DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

使用 @Transactional 注解一个方法使其开启事务:

```
@Service
class UserServiceImpl implements IUserService {

    // 处理删除用户业务逻辑, 使用 @Transactional 注解实现该方法的事务管理
    @Transactional
    public void delUser(long id) {
        ...
    }
}
```

需要注意的是, 开启了事务的方法在数据库错误时应该抛出异常, 否则无法做到事务回滚。

## 4.2.2 Spring Data Redis

基于 Jedis 之上对 Redis 操作的封装。

## 1) 配置 Jedis 连接池。

```
<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
    <property name="maxTotal" value="${redis.pool.maxActive}"/>
    <property name="maxIdle" value="${redis.pool.maxIdle}"/>
    <property name="minIdle" value="4"/>
    <property name="maxWaitMillis" value="${redis.pool.maxWait}"/>
    <property name="testOnBorrow" value="${redis.pool.testOnBorrow}"/>
</bean>
```

## 2) 配置连接工厂。

```
<bean id="jedisConnectionFactory" class="org.springframework.data.redis.
connection.jedis.JedisConnectionFactory">
    <constructor-arg index="0" ref="jedisPoolConfig"/>
    <property name="hostName" value="${redis.host}"/>
    <property name="port" value="${redis.port}"/>
    <property name="usePool" value="true"/>
</bean>
```

## 3) 构造 RedisTemplate 即可使用它来做各种操作。

```
<bean id="redisTemplate" class="org.springframework.data.redis.core.
StringRedisTemplate">
    <property name="connectionFactory" ref="jedisConnectionFactory" />
</bean>
```

```
redisTemplate.opsForValue().set("test_key", "test_value"); //set 操作
redisTemplate.opsForValue().getOperations().delete("test_key"); //del 操作
redisTemplate.opsForHash().put("testKey", "testField", "testValue");
//hset 操作
```

需要注意的是，如果直接使用 RedisTemplate，那么 Redis 的 key 使用的是 String 的 JDK 序列化字节数组，并非是用 String.getBytes() 得到的字节数组。可以通过 RedisTemplate 的 defaultSerializer、keySerializer、valueSerializer、hashKeySerializer 以及 hashValueSerializer 几个属性设置想要使用的序列化机制。其支持的几种序列化机制如下。

- StringRedisSerializer: 简单的字符串序列化，使用的是 String.getBytes() 方法。StringRedisTemplate 就是使用它作为 key、value、hashKey 以及 hashValue 的序列化实现。
- GenericToStringSerializer: 可以将任何对象泛化为字符串并序列化，对于每一种对象类型都有不同的实现。
- JacksonJsonRedisSerializer: JSON 的序列化方式，使用 Jackson Mapper 将 object 序列化为 JSON 字符串。
- Jackson2JsonRedisSerializer: 跟 JacksonJsonRedisSerializer 同样是 JSON 序列化方式，使用的是 jackson databind。

- **JdkSerializationRedisSerializer**: 使用 JDK 自带的序列化机制, 也是直接使用 RedisTemplate 时用的序列化机制。

### 4.2.3 Spring Data MongoDB

基于 Mongo Java Driver 对 MongoDB 的操作封装。使用流程如下。

1) 定义 Mongo 对象并构造数据库工厂, 对应的是 MongoDB 官方 jar 包中的 Mongo, replica-set 设置集群副本的 IP 地址和端口。

```
<!-- -->
<mongo:mongo id="mongo" replica-set="${mongo.hostport}">
  <!-- 一些连接属性的设置 -->
  <mongo:options
    connections-per-host="${mongo.connectionsPerHost}"
    threads-allowed-to-block-for-connection-multiplier="${mongo.threadsAllowedToBlockForConnectionMultiplier}"
    connect-timeout="${mongo.connectTimeout}"
    max-wait-time="${mongo.maxWaitTime}"
    auto-connect-retry="${mongo.autoConnectRetry}"
    socket-keep-alive="${mongo.socketKeepAlive}"
    socket-timeout="${mongo.socketTimeout}"
    slave-ok="${mongo.slaveOk}"
    write-number="1"
    write-timeout="0"
    write-fsync="true"/>
</mongo:mongo>

<mongo:db-factory dbname="${mongo.dbname}" mongo-ref="mongo"
username="${mongo.username}"
password="${mongo.password}">
```

2) 配置映射相关信息, 包括映射上下文、类型映射、映射转换等。

```
<bean id="mappingContext" class="org.springframework.data.mongodb.core.
mapping.MongoMappingContext"/>

<!-- 默认 MongoDB 类型映射 -->
<bean id="defaultMongoTypeMapper" class="org.springframework.data.
mongodb.core.convert.DefaultMongoTypeMapper">
  <constructor-arg name="typeKey">
    <null/>
  </constructor-arg>
</bean>

<!-- 配置 MongoDB 映射类型 -->
<bean id="mappingMongoConverter" class="org.springframework.data.mongodb.
core.convert.MappingMongoConverter">
  <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
```

```

<constructor-arg name="mappingContext" ref="mappingContext"/>
<property name="typeMapper" ref="defaultMongoTypeMapper"/>
</bean>

```

3) 构造 `MongoTemplate`，即可使用 `mongoTemplate` 进行数据库操作。

```

<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.
MongoTemplate">
    <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
    <constructor-arg name="mongoConverter" ref="mappingMongoConverter"/>
    <property name="writeResultChecking" value="EXCEPTION"/>
    <property name="writeConcern" value="ACKNOWLEDGED"/>
</bean>

```

这里需要注意上面配置 `writeResultChecking` 和 `writeConcern` 是为了安全写入。

使用 `MongoTemplate` 需要给数据库实体类加 `@Document`，并指定集合的名字，如果不加此注解或者没指定 `collection`，那么默认使用类的简单名称首字母小写作为集合的名字：

```

@Document(collection = "test_user")
public class User{
    private String id;

    private String userName;

    private String nickName;

    ...
}

```

```

User user = new User();
user.setName("test");
user.setNickName("测试用户");

```

```
mongoTemplate.insert(user); // 插入数据
```

```

user.setName("test2")
mongoTemplate.save(user); // 保存数据

```

```

Criteria criteria = Criteria.where("name").is("test2"); // 根据 name 查询数据
mongoTemplate.findOne(Query.query(criteria), User.class);

```

这里需要注意如下几点。

- Spring Data MongoDB 会默认在每个 `collection` 中添加 `_class` 字段，用于标识原始来源类型，可以在 `defaultMongoTypeMapper` 将 `typeKey` 设置为 `NULL` 来去掉此字段。
- Spring Data MongoDB 会把实体类中的 `id` 属性转换为 `ID`，因此当使用 Spring Mongo 做了数据操作的时候，如果再使用 Mongo Java Driver 查询，注意一定要用 `_id` 而不是 `id`。



- 实体类中的 ID 如果为空，那么会自动生成 ObjectId 作为 ID。
- 要注意对 MongoDB 安全写的配置，根据业务场景对 MongoTemplate 的 writeResultCheckin 和 writeConcern 配置合适的值。

## 4.3 Spring Boot

本章开始提到过 Spring 现在变得越来越复杂，越来越不好上手。这一点 Spring Source 自己也注意到了，因此推出了 Spring Boot，旨在降低使用 Spring 的门槛。其大大简化了 Spring 的配置工作，并且能够很容易地将应用打包为可独立运行的程序（即不依赖于第三方容器，可以独立以 jar 或者 war 包的形式运行）。其带来的开发效率的提升使得 Spring Boot 被看作至少近 5 年来 Spring 乃至整个 Java 社区最有影响力的项目之一，也被人看作 Java EE 开发的颠覆者。另一方面来说，Spring Boot 也顺应了现在微服务（MicroService）的理念，可以用来构建基于 Spring 框架的可独立部署应用程序。

### 4.3.1 Spring Boot 使用示例

一个简单的 POM 配置示例如下：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.7.RELEASE</version>
</parent>

...

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

使用 spring-boot-starter-parent 作为当前项目的 parent，将 Spring Boot 应用相关的一系列依赖（dependency）、插件（plugins）等配置共享；添加 spring-boot-starter-web 这个依赖，是为了构建一个独立运行的 Web 应用；spring-boot-maven-plugin 用于将 Spring Boot 应用以可执行 jar 包的形式发布出去。

接着可以添加相应的 Controller 实现：

```
@RestController
public class MyController {
    @RequestMapping("/")
    public String hello() {
        return "Hello World!";
    }
}
```

这里的 RestController 是一个复合注解，包括 @Controller 和 @ResponseBody。

最后，要让 Spring Boot 可以独立运行和部署，我们需要一个 Main 方法入口，比如：

```
@SpringBootApplication
public class BootDemo extends SpringBootServletInitializer{
    public static void main(String[] args) throws Exception {
        SpringApplication.run(BootDemo.class, args);
    }
}
```

使用 mvn package 打包后（可以是 jar，也可以是 war），java -jar xx.war/jar 即可运行一个 Web 项目，而之所以继承 SpringBootServletInitializer，是为了能够让打出来的 war 包也放入容器中直接运行，其加载原理在 3.4.4 节的零 XML 配置中介绍过。

从最根本上来说，Spring Boot 就是一些库和插件的集合，屏蔽掉了很多配置加载、打包等自动化工作，其底层还是基于 Spring 的各个组件。

这里需要注意的是，Spring Boot 推崇对项目进行零 XML 配置。但是就笔者看来，相比注解配置，其是糅杂在代码中的，每次更新都需要重新编译，在 XML 这种和代码分离的方式下耦合性和可维护性则显得更为合理一些，而且在配置复杂时也更清晰。因此，采用 Java Config 作为应用和组件扫描（Component Scan）入口，采用 XML 做其他的配置，是一种比较好的方式。此外，当集成外部已有系统的时候，通过 XML 集中明确化配置也是更为合理的一种方式。

### 4.3.2 Spring Boot 的运行原理

Spring Boot 的基础组件之一就是 4.1 节介绍过的一些注解配置，除此之外，它也提供了自己的注释。

## 1) @EnableAutoConfiguration。

这个 Annotation 就是 Java Config 的典型代表，标注了这个 Annotation 的 Java 类会以 Java 代码的形式（对应于 XML 定义的形式）提供一系列的 Bean 定义和实例，结合 AnnotationConfigApplicationContext 和自动扫描功能，就可以构建一个基于 Spring 容器的 Java 应用了。

@EnableAutoConfiguration 的定义信息如下：

```
@Target (ElementType.TYPE)
@Retention (RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import (EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

标注了此注解的类会发生一系列初始化动作，分别如下。

- Spring Boot 扫描到 @EnableAutoConfiguration 注解时，就使用 Spring 框架的 SpringFactoriesLoader 去扫描 classpath 下所有 META-INF/spring.factories 文件的配置信息，其中包括如下一些 callback 接口（在前中后等不同时机执行）。
  - org.springframework.boot.SpringApplicationRunListener。
  - org.springframework.context.ApplicationContextInitializer。
  - org.springframework.context.ApplicationListener。
- 然后 Spring Boot 加载符合当前场景需要的配置类型并供当前或者下一步的流程使用，这里说的场景就是提取以 org.springframework.boot.autoconfigure.EnableAutoConfiguration 作为 key 标志的一系列 Java 配置类，然后将这些 Java 配置类中的 Bean 定义加载到 Spring 容器中。

此外，可以使用 Spring 3.0 系列引入的 @Conditional，通过像 @ConditionalOnClass、@ConditionalOnMissingBean 等具体的类型和条件来进一步筛选通过 SpringFactoriesLoader 加载的类。

## 2) Spring Boot 启动。

每一个 Spring Boot 应用都有一个入口类，在其中定义 main 方法，然后使用 SpringApplication 这个类来加载指定配置并运行 Spring Boot Application。如上面写过的入口类：

```

@SpringBootApplication
public class BootDemo extends SpringBootServletInitializer{
    public static void main(String[] args) throws Exception {
        SpringApplication.run(BootDemo.class, args);
    }
}

```

`@SpringBootApplication` 注解是一个复合注解，包括了 `@Configuraiton`、`@EnableAutoConfiguration` 以及 `@ComponentScan`。通过 `SpringApplication` 的 `run` 方法，Spring 就使用 `BootDemo` 作为 Java 配置类来读取相关配置，加载和扫描相关的 Bean。

这样，基于 `@SpringBootApplication` 注解，Spring 容器会自动完成指定语义的一系列工作，包括 `@EnableAutoConfiguration` 要求的東西，如从 Spring Boot 提供的多个 starter 模块中加载 Java Config 配置，然后将这些 Java Config 配置筛选上来的 Bean 定义加入 Spring 容器中，再 refresh 容器。一个 Spring Boot 应用即启动完成。

### 4.3.3 Spring Boot 的组成模块

Spring Boot 是由非常多的模块组成的，可以通过 POM 文件引入进来。`EnableAutoConfiguration` 机制会进行插件化加载并自动配置，这里模块化机制的原理主要是通过判断相应的类 / 文件是否存在来实现的。其中几个主要的模块如下。

- `spring-boot-starter-web`

此模块用于标记此项目是一个 Web 应用，Spring Boot 会自动准备好相关的依赖和配置。

这里 Spring Boot 默认使用 Tomcat 作为嵌入式 Web 容器，可以通过声明 `spring-boot-starter-jetty` 的 dependency 来换成 Jetty。

- `spring-boot-starter-logging`

Spring Boot 对此项目开启 SLF4J 和 Logback 日志支持。

- `spring-boot-starter-redis`

Spring Boot 对此项目开启 Redis 相关依赖和配置来做数据存储。

- `spring-boot-starter-jdbc`

Spring Boot 对此项目开启 JDBC 操作相关依赖和配置来做数据存储。

这里需要说明的是，Spring Boot 提供的功能非常丰富，因此显得笨重、复杂。其实依赖于模块插件化机制，我们可以只配置自己需要的功能，从而对应用进行瘦身，避免无用的配置影响应用启动速度。

### 4.3.4 小结

Spring Boot 为使用 Spring 做后端应用开发带来了非常大的便利，能够大大提高搭建应用雏形框架的速度，而大家只需要关注实现业务逻辑。其“黑魔法”一样的插件化机制提供了非常好的灵活性，使得能够根据自己的需要引入所需的组件。如果非遗留 Spring 项目，直接使用 Spring Boot 是比较好的选择；遗留项目也可以通过配置达到无缝结合。

## 4.4 Spring 常用组件

除了前几节讲述的 Spring 核心组件和数据操作组件外，Spring 还有一些常用的组件。

### 4.4.1 表达式引擎——Spring Expression Language

Spring 3.0 引入了 Spring Expression Language (SpEL)，这是一种在运行时给 Bean 的属性或者构造函数参数注入值的方法。其具有以下优点。

- 可以通过 Bean 的 ID 引用 Bean。
- 可以调用某个对象的方法或者访问它的属性。
- 支持数学、关系和逻辑操作。
- 正则表达式匹配。
- 支持集合操作。

SpEL 表达式被 `#{}`  包围，跟 placeholders 中的 `#{}`  非常像，最简单的 SpEL 表达式可以写作 `#{}` 。以下是几个例子。

- `#{}(T(System).currentTimeMillis())`：这个表达式负责获取当前的系统时间，`T()` 操作符负责将 `java.lang.System` 解析成类，以便可以调用 `currentTimeMillis()` 方法或者访问其静态变量。
- `#{}(systemProperties['spring.profile'])`：引用系统属性 `spring.profile`。
- `#{}(adminUser.userName)`：引用 ID 为 `adminUser` 的属性 `userName`。
- `#{}(adminUser.getUserName()?.toUpperCase())`：调用 `adminUser` 的 `getUserName` 方法，这里的 `?` 的作用是防止 `getUserName` 的返回值为 `NULL`。
- `#{}(adminUser.noteList.[title])`：把 `adminUser` 的 `noteList` 中的每一个 `note` 的 `title` 都提取出来构成一个新的字符串集合。

引用 SpEL 的这些值可以通过 @Value 注解。

```
public UserService(" class="java.lang.String")

    @Value("#{adminUser.userName}")
    private String adminUserName;

    ...
}
```

此外，也可以使用代码做 EL 解析和执行。

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationConte
xt();
ctx.refresh();
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setBeanResolver(new BeanFactoryResolver(ctx));
//context.registerFunction(HASH, hash); // 注册方法
//context.setVariable(ARGS, arguments); // 注册变量
Properties result = parser.parseExpression("@systemProperties").
getValue(context, Properties.class);
Assert.assertEquals(System.getProperties(), result);
```

## 4.4.2 远程过程访问的支持——Spring Remoting

Spring Web 提供了几种远程服务的封装，基本都用一个 ServiceExporter 发布服务，一个 ProxyFactoryBean 引用服务。

- RMI：对 Java 自带 RMI 机制的封装，包括 RmiServiceExporter 和 RmiProxyFactoryBean。
- Spring HTTP Invoker：Spring 自己实现的一种基于 HTTP 协议且使用 Java 的序列化机制的 RPC 方式，包括 HttpInvokerProxyFactoryBean 和 HttpInvokerServiceExporter。
- Hessian：Hessian 协议的封装使用，包括 HessianProxyFactoryBean 和 HessianServiceExporter。
- Burlap：Burlap 是一种基于 XML 的 RPC 协议，包括 BurlapProxyFactoryBean 和 BurlapServiceExporter。

这些远程服务封装的使用方法，基本是类似的，如图 4-5 所示。

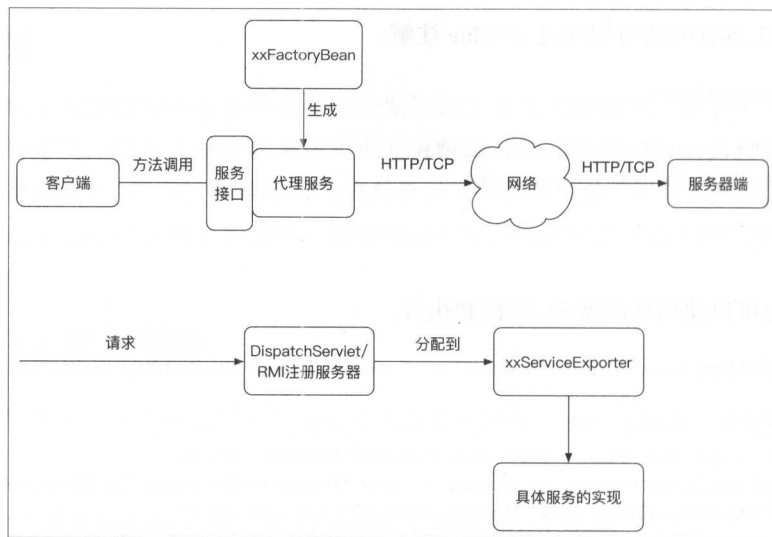


图 4-5

### 4.4.3 Spring 与 JMX 的集成

Java 管理扩展（Java Management Extension, JMX）从 Java 5.0 开始引入，提供连接、监控和管理远程 JVM 的方式。如果一个 Java 对象可以由一个遵循 JMX 规范的管理器应用管理，那么这个 Java 对象就可以视为一个可由 JMX 管理的资源。使用 Spring 与 JMX 集成，实现方式灵活而且简单。

- 可以自动探测实现 MBean 接口的 MBean 对象，而且可以将一个普通的 Spring Bean 注册为 MBean。
- 定制管理 MBean 的接口，根据需要暴露特定管理 MBean 的操作。
- 使用注解定义 MBean 管理接口。
- 可以实现对本地和远程 MBean 的代理。

声明 MBean：

```
public interface AdminMBean {
    public void invoke(String className, String methodName, Object[] args);
}

public class Admin implements AdminMBean {
    public void invoke(String className, String methodName, Object[] args) {
        ...
    }
}
```

配置并暴露服务：

```
<bean id="adminBean" class="me.rowkey.pje.jmx.impl.Admin" />

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter"
    lazy-init="false">
    <property name="beans">
        <map>
            <entry key="me.rowkey.pje.jmx.impl:name=AdminBean"
                value-ref="adminBean" />
        </map>
    </property>
</bean>

<bean id="registry" class="org.springframework.remoting.rmi.
RmiRegistryFactoryBean">
    <property name="port" value="9999" />
</bean>

<bean id="serverConnector"
    class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=webapp.rmi" />
    <property name="serviceUrl"
        value="service:jmx:rmi:///jndi/rmi://localhost:9999}/
kmconnector" />
</bean>
```

以上便在 9999 端口开放了 JMX 协议并发布了一个 MBean，其底层通信是基于 RMI 的。这样通过 serviceUrl: "service:jmx:rmi:///jndi/rmi://127.0.0.1:9099/kmconnector" 和 mbeanName: "suishen.libs.admin.jmx.impl:name=AdminBean" 即可调用服务。

#### 4.4.4 定时任务的支持——Spring Quartz

Spring Quartz 是 Spring 自己实现的一套定时调度框架，支持 Cron 表达式。需要注意的是，相比真正的 Quartz 框架，Spring Quartz 是轻量级的，缺乏分布式等高级特性。

典型的使用 Spring Quartz 的 XML 配置如下：

```
<task:scheduler id="quartzScheduler" pool-size="10"/>

<task:scheduled-tasks scheduler="quartzScheduler">

    <task:scheduled ref="userDisableService" method="enableService"
cron="* * /5 * * * ?"/>

</task:scheduled-tasks>
```

其中 task:scheduler 配置 Quartz 使用的 scheduler 的线程池大小为 10。配置的任务则是每 5 分钟执行 userDisableService 这个 Bean 的 enableService 方法。



当然，Spring Quartz 也支持注解配置，开关如下：

```
<task:annotation-driven scheduler="qbScheduler" mode="proxy"/>
```

当然，也可以使用 @ 注解 EnableScheduling 来开启 Spring Quartz 的注解支持。

这样就可以在 Bean 中使用 @Scheduled 注解来配置定时任务了：

```
@Service
public class UserDisableService{

    @Scheduled(cron = "* */5 * * * ?")
    public void enableService(){
        ...
    }
}
```

#### 4.4.5 跨域请求的支持——Spring CORS

CORS ( Cross-Origin Resource Sharing ) 用于解决浏览器中跨域请求的问题。简单的 Get 请求可以使用 JSONP 解决，而对于其他稍微复杂的请求则需要后端应用支持 CORS。Spring 4.2 之后提供了 @CrossOrigin 注解实现对 CORS 的支持。

- 在 Controller 方法上配置。

```
@CrossOrigin(origins = {"http://localhost:8088"})
@RequestMapping(value = "/corsTest", method = RequestMethod.GET)
public String greetings() {
    return "cors test";
}
```

- 在 Controller 上配置，那么此 Controller 中所有的 method 都支持 CORS。

```
@CrossOrigin(origins = "http://localhost:8088", maxAge = 3600)
@Controller
@RequestMapping("/api/")
public class TestController {
    @RequestMapping(value = "/corsTest", method = RequestMethod.
    GET)
    public String greetings() {
        return "cors test";
    }
}
```

- Java Config 全局配置。

```
@Configuration
@EnableWebMvc
public class SpringWebConfig extends WebMvcConfigurerAdapter {
    @Override
```

```

public void addCorsMappings(CorsRegistry registry) {
    // 对所有 URL 配置
    //registry.addMapping("/*");

    // 针对某些 URL 配置
    registry.addMapping("/api/*").allowedOrigins("http://
localhost:8888")
        .allowedMethods("PUT", "DELETE")
        .allowedHeaders("header1", "header2", "header3")
        .exposedHeaders("header1", "header2")
        .allowCredentials(false).maxAge(3600);
}
}

```

- XML 全局配置。

```

<mvc:cors>
    <!--<mvc:mapping path="/*" />-->

    <mvc:mapping path="/api/*" allowed-origins="http://
localhost:8888, http://localhost:8088" allowed-methods="GET, PUT"
        allowed-headers="header1, header2, header3" exposed-
headers="header1, header2"
        allow-credentials="false" max-age="123"/>
</mvc:cors>

```

## 4.5 总结

本章讲述了 Spring Core 以及几个常用组件的使用。除此之外 Spring 中还有其他非常多的组件都提供了各个方面的功能封装，如下。

- Spring Security：安全、权限控制。
- Spring Retry：重试机制的封装框架。
- Spring OAuth：OAuth 实现框架。
- Spring JMS：JMS 的封装。
- Spring AMQP：AMQP 的使用封装框架。
- Spring Rabbit：RabbitMQ 的使用封装框架。

额外地，这里对目前非常火热的 Spring Cloud 做一下简单介绍。

Spring Cloud 给我们构建分布式系统提供了一整套开发工具和框架。现在很多公司和团队都基于 Spring Cloud 这一套东西在做微服务实现。不过，Spring Cloud 包含很多子项目，想要吃透这些得花不小的成本。

Spring Cloud 的主要子项目介绍如下。

- **Spring Cloud Config**: 统一配置中心, 类似于前文说过的 Disconf, 不过其配置文件是存储在版本管理系统 (如 Git、SVN) 上的。其配置的实时在线更新则需要依赖 Spring Cloud Bus。
- **Spring Cloud Security**: 提供了 OAuth 2 客户端的负载均衡以及认证 header 等安全服务, 可以作为 API 网关的实现。
- **Spring Cloud Consul/Zookeeper**: 服务统一发现、注册、配置, 类似于 Dubbo。
- **Spring Cloud Bus**: 提供了服务之间通信的分布式消息事件总线, 主要用来在集群中传播状态改变 (如配置改动)。
- **Spring Cloud Sleuth**: 分布式追踪系统, 能够追踪单次请求的链路轨迹以及耗时等信息。

此外, 使用 Spring Cloud Netflix 则能够集成使用 Netflix 的各个组件构建服务。Netflix 的主要组件如下。

- **Zuul**: 这是 Netflix 所有后端服务最前端的一道门, 也就是我们上面说的 API 网关, 主要包含了以下功能。
  - **认证授权和安全**: 识别合法的外部请求, 拒绝非法的。
  - **监控**: 追踪记录所有有意义的数据以便给我们一个精确的产品视图。
  - **动态路由**: 根据需要动态地把请求路由到合适的后端服务上。
  - **压力测试**: 渐进式地增加对集群的压力直到最大值。
  - **限流**: 对每一种类型的请求都限定流量, 拒绝超出的请求。
  - **静态响应控制**: 对于某些请求直接在边缘返回而不转发到后端集群。
  - **多区域弹性**: 在 AWS 的多个 Region 中进行请求路由。
- **Eureka**: 是 Netflix 的服务注册发现服务, 类似于 Dubbo 的功能, 包括负载均衡和容错。
- **Hystrix**: Hystrix 是一个类库。基于命令模式, 实现了依赖服务的容错、降级、隔离等。在依赖多个第三方服务的时候非常有用。此外, 还可以通过自定义实现 Dubbo 的 filter 来给 Dubbo 添加 Hystrix 的特性支持。

# 第 5 章

## 数据存储

在应用程序中，很多数据，如用户的信息、商品的信息等都是需要保存的。临时存储在内存中、写入文件中、存储到数据库、存储到搜索引擎，都是数据存储的方式。数据存储可以说是应用开发中处于最基础位置的组件。

对于数据存储中最常用的数据库来说，可以从存储介质上分为如下两种。

- **内存数据库：**数据主要存储在内存中，同时也可以将数据持久化存储到硬盘中，如 Redis、H2DB 的内存模式。对于这种数据库，由于内存成本昂贵，因此一定要做好存储的量化分析、容量预估，防止内存不足造成服务不可用。
- **硬盘数据库：**数据存储在硬盘上的这种数据库是最常见的。MySQL、Oracle、PostgreSQL、HBase、H2DB、SQLite 等都是硬盘数据库。此外，SSDB 是基于 SSD 硬盘的 KV 数据库，支持的数据接口很丰富，是 Redis 的另外一个选择。

这里暂且抛开文件存储以及存储介质不谈，主要针对包括数据库在内的以下几种常用存储方式进行讲述。

- **关系型数据库：**关系型数据库是用得最普遍的数据库，也是在计算机课堂上讲得最多的数据库。典型的关系型数据库包括 MySQL、Oracle、PostgreSQL 等。
- **非关系型数据库：**非关系型数据库是最近这些年兴起的数据库，区别于关系型数据库，其并不遵循关系模型，因此使用场景和关系型数据库是不同的。非关系型数据库以 MongoDB、CouchDB、HBase 为代表。
- **缓存系统：**缓存是为了解决传统文件读 / 写性能缓慢的问题而产生的主要以内存为存储介质的软件，一般是数据库的辅助部分，并不要求持久化存储。缓存系统以 Memcached、Redis 为代表。

- **搜索引擎**：为搜索场景而诞生的软件，一般基于倒排索引实现，能够提供很好的搜索性能。搜索引擎以 Solr、Elasticsearch 为代表。

## 5.1 关系型数据库——MySQL

关系型数据库是采用关系模型来组织数据的数据库。所谓关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表格及其之间的联系所组成的一个数据组。目前，用得最普遍的关系型数据库有 MySQL、Oracle、PostgreSQL 等。这里针对 Java 开发中最常用的 MySQL 进行阐述。MySQL 版本为 5.5.19。

### 5.1.1 存储引擎

MySQL 主要有两种存储引擎，分别是 MyISAM 和 InnoDB。

#### 1) MyISAM

MySQL 5.5 之前的默认引擎，特点如下。

- 不支持行锁，读取时对需要读到的所有表加锁，写入时则对表加排他锁。
- 不支持事务。
- 不支持外键。
- 不支持崩溃后的安全恢复。
- 在表有读取查询的同时，支持往表中插入新记录。
- 支持 BLOB 和 TEXT 的前 500 个字符索引，支持全文索引。
- 支持延迟更新索引，极大地提升了写入性能。
- 对于不会进行修改的表，支持压缩表，极大地减少了磁盘空间占用。

#### 2) InnoDB

MySQL 5.5 后的默认引擎，特点如下。

- 支持行锁，采用 MVCC 来支持高并发，有可能死锁。
- 支持事务。
- 支持外键。

- 支持崩溃后的安全恢复。
- 不支持全文索引。

由于 MyISAM 缓存有表 meta-data (行数等), 因此在做 COUNT(\*) 时对于一个结构很好的查询是不需要消耗多少资源的。而对于 InnoDB 来说, 则没有这种缓存。当你需要行锁定、事务时, 使用 InnoDB 则是更好的选择, 也具有更高级的安全性。此外, MyISAM 和 InnoDB 使用的索引也是有区别的, 前者为非聚簇索引, 后者为聚簇索引。

总体来说, MyISAM 适合读密集型的表, 而 InnoDB 适合写密集型的表。在数据库做主从分离的情况下, 经常选择 MyISAM 引擎作为主库的存储引擎。

## 5.1.2 字符集和校对规则

在创建数据库、数据表的时候会指定 character 和 collate。如下:

```
CREATE DATABASE db_name DEFAULT CHARSET SET utf8 COLLATE utf8_general_ci;

CREATE TABLE `note` (
  `id` bigint(11) unsigned NOT NULL AUTO_INCREMENT,
  `content` varchar(128) CHARSET utf8 COLLATE utf8_bin NOT NULL COMMENT '内容',
  // 对某一个字段设置字符集和校对规则
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8 COLLATE=utf8_general_ci;
```

字符集指的是一种从二进制编码到某类字符符号的映射。校对规则则是指某种字符集下的排序规则。MySQL 中每一种字符集都会对应一系列的校对规则。

MySQL 采用类似继承的方式来设定字符集默认值, 每个数据库以及每张数据表都有自己的默认值, 它们逐层继承, 最终最靠底层的默认设置将影响你创建的对象。比如, 在创建数据库时, 将根据服务器上的 character\_set\_server 来设置数据库的默认字符集, 同样的道理, 根据 database 的字符集来指定库中所有表的字符集。不管是对数据库, 还是对表和列, 只有在它们没有显式指定字符集的情况下, 默认字符集才会起作用。

MySQL 提供了很多字符集供使用, 应该根据存储的内容选择能满足需求的最小字符集。如果存储的内容是英文字符等拉丁语系字符的话, 那么使用默认的 latin1 编码即可; 如果需要存储汉字、俄文、阿拉伯语等非拉丁语系字符, 则建议使用 UTF8 字符集。而校对规则一般来说只需要考虑是否以大小写敏感的方式比较字符串或者是否用字符串编码的二进制形式来比较大小, 其对应的校对规则的后缀分别是 \_cs、\_ci 和 \_bin。大小写敏感和二进制校对规则的不同之处在于, 二进制校对规则直接使用字符的字节进行比较, 而大小写敏

感的校对规则在多字节字符集情况下，如德语，有更复杂的比较规则。一般我们选择大小写不敏感的校对规则即可。

UTF8 是我们常用的字符集，其对应的常用校对规则如下。

- `utf8_general_ci`: UTF8 的默认校对规则。这是一个遗留的校对规则，不支持扩展，仅能够在字符之间进行逐个比较，因此比较速度很快。
- `utf8_unicode_ci`: 根据 Unicode 校对规则算法（UCA）执行，其最主要的特色是支持扩展，即可以把一个字母看作与其他字母组合相等。例如，在德语和一些其他语言中“ß”等于“ss”。但其仅部分支持 Unicode 校对规则算法，仍有一些字符是不能支持的，并且不能完全支持组合的记号。其正确性要比 `utf8_general_ci` 高。
- `utf8_bin`: 将字符串每个字符用二进制数据编译存储，区分大小写，而且可以存二进制的內容。

通常情况下，`utf8_general_ci` 的准确性足够我们使用。

此外，如果数据库需要支持 emoji 字符，那么字符集应该用 UTF8MB4，校对规则使用 `utf8mb4_bin`（如果使用 `utf8mb4_general_ci` 的话，会出现某些 emoji 表情无法区分的问题）。UTF8MB4 是 MySQL 在 5.5.3 之后增加的编码，MB4 就是 Most Bytes 4 的意思，专门用来兼容 4 字节的 Unicode。它是 UTF8 的超集，除了将编码改为 UTF8MB4 外不需要做其他转换。笔者建议，为了兼容性，使用 UTF8MB4 编码。

### 5.1.3 索引的使用

数据库索引是数据库使用中非常关键的技术，合理正确地使用索引能够大大提高数据库的查询性能。

#### 数据结构

MySQL 索引使用的数据结构主要有 BTree 索引和哈希索引。对于哈希索引来说，底层数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择使用哈希索引，查询性能最快；其余大部分索引场景，则建议选择 BTree 索引。

MySQL 的 BTree 索引使用的是 B 树中的 B+Tree，但对于主要的两种存储引擎其实现方式是不同的。

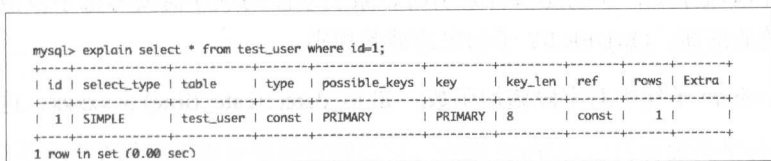
- **MyISAM**: B+Tree 叶节点的 data 域存放的是数据记录的地址。在索引检索时, 首先按照 B+Tree 搜索算法搜索索引, 如果指定的 Key 存在, 则取出其 data 域的值, 然后以 data 域的值为地址读取相应的数据记录。这被称为非聚簇索引。
- **InnoDB**: 其数据文件本身就是索引文件。相比 MyISAM, 索引文件和数据文件是分离的, 其表数据文件本身就是按 B+Tree 组织的一个索引结构, 树的叶节点 data 域保存了完整的数据记录。这个索引的 key 是数据表的主键, 因此 InnoDB 表数据文件本身就是主索引。这被称为聚簇索引 (也叫聚集索引)。而其余的索引都为辅助索引, 辅助索引 data 域存储相应记录主键的值而不是地址, 这也是和 MyISAM 不同的地方。在根据主索引搜索时, 直接找到 key 所在的节点即可取出数据; 在根据辅助索引查找时, 则需要先取出主键的值, 再走一遍主索引。因此, 在设计表的时候, 不建议使用过长的字段作为主键, 也不建议使用非单调的字段作为主键, 这样会造成主索引频繁分裂。

## 索引使用

先介绍一个常用的 MySQL 命令 `explain`, 此命令能够打印出 SQL 语句的执行计划, 从而判断要执行的 SQL 语句是否能够命中索引, 并做进一步调整。在优化 SQL 查询时, 最好的方式就是使用此命令来判断执行计划是否合理。如下:

```
explain select * from test_user where id=1;
```

结果如图 5-1 所示。



```
mysql> explain select * from test_user where id=1;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	test_user	const	PRIMARY	PRIMARY	8	const	1	

1 row in set (0.00 sec)

图 5-1

这里需要注意的是 `type`、`key` 和 `extra` 这 3 列, 分别介绍如下。

- **type**: 显示了连接使用了哪种类别, 有无使用索引。如果为 `ALL`, 那么说明要进行全表扫描。
- **key**: 此列显示 MySQL 实际决定使用的键 (索引)。如果没有选择索引, 键是 `NULL`。要想强制 MySQL 使用或忽视 `possible_keys` 列中的索引, 则在查询中使用 `FORCE INDEX`、`USE INDEX` 或者 `IGNORE INDEX`。如果这里为 `NULL`, 则说明没有命中索引。



- Extra: 此列如果出现 Using filesort (需要额外的步骤来发现如何对返回的行排序) 或者 Using temporary (需要创建一个临时表来存储结果), 说明查询需要优化。

这里需要说明的是, MySQL 自身是有查询优化器的, 优化器的作用就是在一个查询所有可能的执行方式中找到其中最好的执行计划。因此 explain 获取到的执行计划并非是固定的, 它会随着数据分布情况而变动, 执行计划也有可能改变。而且当数据库计算出使用索引所耗费的时间长于全表扫描或其他操作时 (比如当表中索引字段数据重复率太高), 将不会使用索引。

#### 1) 最左前缀原则。

MySQL 中的索引可以以一定顺序引用多列, 这种索引叫作联合索引。如 User 表的 name 和 city 加联合索引就是 (name,city)。而最左前缀原则指的是, 如果查询的时候查询条件精确匹配索引的左边连续一列或几列, 则此列就可以被用到。如下:

```
select * from user where name=xx and city=xx; // 可以命中索引
```

```
select * from user where name=xx; // 可以命中索引
```

```
select * from user where city=xx; // 无法命中索引
```

这里需要注意的是, 查询的时候如果两个条件都用上了, 但是顺序不同, 如 city = xx and name = xx, 那么现在的查询引擎会自动优化为匹配联合索引的顺序, 这样是能够命中索引的。

由于最左前缀原则, 在创建联合索引时, 索引字段的顺序需要考虑字段值去重之后的个数, 较多的放前面。ORDER BY 子句也遵循此规则。

2) 避免 where 子句中对字段施加函数, 如 to\_date(create\_time) > xxxxxx, 这会造成无法命中索引。

3) 在使用 InnoDB 时使用与业务无关的自增主键作为主键, 即使用逻辑主键, 而不要使用业务主键。

#### 4) 合理利用索引覆盖。

覆盖索引 (covering index) 指一个查询语句的执行只需要从辅助索引中就可以得到查询记录, 而不需要查询聚集索引中的记录, 也可以称之为实现了索引覆盖。简单来说就是查询条件命中了索引, 而查询字段也属于索引中的字段, 那么就实现了索引覆盖。当实现覆盖索引的时候, explain 命令的 Extra 会显示 using Index。

## 5) 避免冗余索引。

冗余索引指的是索引的功能相同, 能够命中 A 就肯定能命中 B, 那么 A 就是冗余索引。如 (name,city) 和 (name) 这两个索引就是冗余索引, 能够命中后者的查询肯定是能够命中前者的。在大多数情况下, 都应该尽量扩展已有的索引而不是创建新索引。

MySQL 5.7 版本后, 可以通过查询 sys 库的 schemal\_redundant\_indexes 表来查看冗余索引。

6) 将打算加索引的列设置为 NOT NULL, 否则将导致引擎放弃使用索引而进行全表扫描。

7) 删除长期未使用的索引, 不用的索引的存在会造成不必要的性能损耗。MySQL 5.7 后可以通过查询 sys 库的 schema\_unused\_indexes 视图来查询哪些索引从未被使用。

8) 联表查询必须存在的情况下, 可以使用索引提高性能。

联表的索引使用要注意如下两点。

- 确保 ON 和 USING 中的列上有索引。在创建索引的时候就要考虑关联的顺序。当表 A 和表 B 用列 c 关联的时候, 如果优化器关联的顺序是 A、B, 只需要在 B 的 c 字段建立索引即可。
- 确保任何的 GROUP BY 和 ORDER BY 中的表达式只涉及一个表中的列, 这样 MySQL 才有可能使用索引来优化。

9) 在使用 limit offset 查询缓慢时, 可以借助索引来提高性能:

```
SELECT * FROM test_user a JOIN (select id from test_user limit ?, ?) b
    ON a.id = b.id
    order by create_time desc
```

10) 查询条件的字段应使用正确的数据类型, 否则 MySQL 会自动做类型转换, 导致无法命中索引。例如 test\_user 表中 mobile 列为字符串类型, 查询的时候如果没有加'', 那么就会进行强制类型转换。

索引可以加快查询速度, 但索引也是有代价的: 索引文件本身要消耗存储空间, 并且在被索引的表上 INSERT 和 DELETE 会变慢; 另外, MySQL 在运行时也要消耗资源维护索引, 因此索引并不是越多越好。在如下两种情况下不建议建索引。

- 表记录比较少, 例如一两千条甚至只有几百条记录的表, 没必要建索引。
- 索引的选择性较低。所谓索引的选择性 (Selectivity), 是指不重复的索引值 (也叫基数, Cardinality) 与表记录数 (#T) 的比值:

$$\text{Index Selectivity} = \text{Cardinality} / \#T$$

显然选择性的取值范围为 (0, 1], 选择性越高的索引价值越大, 这是由 B+Tree 的性质决定的。在 MySQL 5.6 后, MySQL 库下的 innodb\_index\_stats 表的 stat\_value 字段记录了某张表在某个索引的不同取值的记录个数, innodb\_table\_stats 的 n\_rows 字段记录了某张表总的记录数目, 两者相除即为索引的区分度。

此外, 需要提到的是 MySQL 也支持全文索引 (5.6.24 之前 MyISAM 引擎支持, 之后 InnoDB 也开始支持):

```
CREATE TABLE user_note (
  id INT AUTO_INCREMENT NOT NULL PRIMARY KEY,
  title VARCHAR(200),
  FULLTEXT(title)
) TYPE=MYISAM;
```

```
SELECT * FROM `user_note` WHERE MATCH(`title`) AGAINST(' 篮球 ')
```

### 5.1.4 查询缓存的使用

my.cnf 加入以下配置, 重启 MySQL 开启查询缓存:

```
query_cache_type = 1
query_cache_size = 600000
```

MySQL 命令行执行以下命令, 也可开启查询缓存:

```
set global query_cache_type = 1;
set global query_cache_size = 600000;
```

如上, 开启查询缓存后在同样的查询条件以及数据情况下, 会直接在缓存中返回结果。这里的查询条件包括查询本身、当前要查询的数据库、客户端协议版本号等一些可能影响结果的信息。因此任何两个查询在任何字符上的不同都会导致缓存不命中。此外, 如果查询中包含任何用户自定义函数、存储函数、用户变量、临时表、MySQL 库中的系统表, 其查询结果也不会被缓存。

缓存建立之后, MySQL 的查询缓存系统会跟踪查询中涉及的每张表, 如果这些表 (数据或结构) 发生变化, 那么和这张表相关的所有缓存数据都将失效。

缓存虽然能够提升数据库的查询性能, 但是缓存也同时带来了额外的开销, 每次查询后都要做一次缓存操作, 失效后还要销毁。因此, 开启缓存查询要慎重, 尤其对于写密集的应用来说更是如此。如果开启, 要注意合理控制缓存空间大小, 一般来说其大小设置为几十 MB 比较合适。此外, 还可以通过 sql\_cache 和 sql\_no\_cache 来控制某个查询语句是否需要缓存:

```
select sql_no_cache count(*) from test_user;
```

### 5.1.5 数据同步中的 Binlog

MySQL 的 Binlog 是用来做 POINT-IN-TIME 的恢复和主从复制的, 由数据库上层生成, 是 SQL 执行的逻辑日志, 在事务提交完成后进行一次写入。

Binlog 有如下 3 种格式。

#### 1) Statement

MySQL 的默认 Binlog 格式。每一条会修改数据的 SQL 都被记录到 bin-log 中, Slave 在复制的时候 SQL 进程会解析成和原来 Master 端执行过的相同的 SQL 后再次执行。

此格式产生的日志量比较少, 能够节省 I/O 和存储资源, 日志具有较高的可阅读性。但其需要在记录语句信息的同时也记录语句执行时的上下文信息, 以保证在 Slave 端重放时能够得到和 Master 端同样的结果。此外, 并非所有的 UPDATE 语句都能够被复制, 尤其是在包含不确定操作的时候, 并且 Slave 的数据表必须和 Master 的保持一致。

#### 2) Row

日志中会记录每一行数据被修改的形式, 然后在 Slave 端再对相同的数据进行修改。其不需要记录语句执行的上下文信息, 但是需要记录每一条记录的改动, 因此当受影响的记录很多时, 日志量会非常大; 由于加密, 日志的可阅读性较低。

#### 3) Mixed

Statement 和 Row 的结合。会根据执行的每一条具体的 SQL 语句来区别对待记录的日志形式, 也就是在 Statement 和 Row 之间选择一种。

### 5.1.6 事务机制

关系型数据库是需要遵循 ACID 规则的, 分别介绍如下。

- A (Atomic) 原子性: 即事务要么全部做完, 要么全部都不做。只要其中一个操作失败, 就认为事务失败, 需要回滚。
- C (Consistency) 一致性: 数据库要一直处于一致的状态。
- I (Isolation) 独立性: 并发的事务之间不会互相影响。
- D (Durability) 持久性: 一旦事务提交后, 它所做的修改将会永久地保存在数据库中。

为了达到以上事务特性, 数据库定义了几种事务隔离级别。

1) 未授权读取 (Read Uncommitted) : 会产生脏读, 可以读取未提交的记录, 实际情况下不会使用。

2) 授权读取 (Read Committed) : 会存在不可重复读以及幻读的现象。不可重复读重点在修改, 即读取过的数据两次读的值不一样; 幻读则侧重于记录数目变化, 多次执行同一个查询返回的记录不完全相同。

3) 可重复读取 (Repeatable Read) : 解决了不可重复读的问题, 会存在幻读现象。InnoDB 使用 MVCC+Gap Lock (InnoDB 行锁的一种) 避免了幻读问题。

4) 串行 (Serializable) : 也称可串行读, 此级别下读操作会隐式获取共享锁, 保证不同事务间的互斥。其消除了脏读、幻读, 但事务并发度急剧下降。

这里需要注意的是, MySQL 的默认事务级别为 Repeatable Read, 而 JDBC 的默认事务级别为 Read Committed, 因此使用的时候要特别注意。此外, 由于 Read Committed 有不可重复读的问题, 因此不能在 Statement 格式的 Binlog 下使用, 必须设置为 Mixed 或者 Row。

事务隔离的实现基于锁机制和并发调度。其中并发调度使用的是 MVCC (多版本并发控制), 通过保存修改行的旧版本信息来支持并发一致性读和回滚等特性。

## 锁机制

MySQL 为了解决并发、数据安全的问题, 使用了锁机制。

可以按照锁的粒度把数据库锁分为表级锁和行级锁。

- **表级锁**: 是 MySQL 中锁定粒度最大的一种锁, 对当前操作的整张表加锁, 实现简单, 资源消耗较少, 加锁快, 不会出现死锁。其锁定粒度最大, 触发锁冲突的概率最高, 并发度最低。MyISAM 和 InnoDB 引擎都支持表级锁。
- **行级锁**: 是 MySQL 中锁定粒度最小的一种锁, 只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小, 并发度高, 但加锁的开销也最大, 加锁慢, 会出现死锁。InnoDB 支持的行级锁, 包括如下这几种。
  - **Record Lock** 对索引项加锁, 锁定符合条件的行。其他事务不能修改和删除加锁项。
  - **Gap Lock**: 对索引项之间的“间隙”加锁, 锁定记录的范围 (对第一条记录前的间隙或最后一条记录后的间隙加锁), 不包含索引项本身。其他事务不能在锁范围内插入数据, 这样就防止了别的事务新增幻影行。
  - **Next-key Lock**: 锁定索引项本身和索引范围。即 Record Lock 和 Gap Lock 的结合, 可解决幻读问题。

虽然使用行级锁具有粒度小、并发度高等优点，但是表级锁有时候也是有必要的：

- 事务更新大表中的大部分数据直接使用表级锁效率会更高。
- 事务比较复杂，使用行级锁很可能引起死锁导致回滚。

表级锁和行级锁可进一步划分为共享锁和排他锁，分别介绍如下。

- **共享锁 (S)**：又被称为读锁，是读取操作创建的锁。其他用户可以读取数据，可以再加共享锁，读取到的数据也是同一版本的；但任何事务都不能获取数据上的排他锁，不能对数据进行修改。获取共享锁的事务只能读取数据而不能修改数据。可以使用 `SELECT ... LOCK IN SHARE MODE` 来强制获取共享锁，否则绝大部分查询操作是不会获取锁的（串行事务级别除外）。
- **排他锁 (X)**：又被称为写锁，一个事务对数据加上排他锁后，其他事务不能再对此数据加任何其他类型的锁。获取排他锁的事务既能读取数据也能修改数据。InnoDB 对 CUD (insert、update、delete) 操作涉及的数据会默认加排他锁。对于查询语句可以使用 `SELECT ... FOR UPDATE` 加排他锁。

InnoDB 中还有如下两个表级锁。

- **意向共享锁 (IS)**：表示事务准备给数据行加入共享锁，事务在一个数据行加共享锁前必须先取得该表的 IS 锁。
- **意向排他锁 (IX)**：表示事务准备给数据行加入排他锁，事务在一个数据行加排他锁前必须先取得该表的 IX 锁。

这里的意向锁是表级锁，表示的是一种意向，仅仅表示事务正在读或写某一行记录，在真正加行级锁时才会判断是否冲突。意向锁是 InnoDB 自动加的，不需要用户干预。

InnoDB 的锁机制兼容表如表 5-1 所示。

表 5-1

当前锁模式 \ 请求锁模式	X	IX	S	IS
X	冲突	冲突	冲突	冲突
IX	冲突	兼容	冲突	兼容
S	冲突	冲突	兼容	兼容
IS	冲突	兼容	兼容	兼容

当一个事务请求的锁模式与当前的锁兼容，InnoDB 就将请求的锁授予该事务；反之如果请求不兼容，则该事务就等待锁释放。

需要注意的是, InnoDB 的行级锁是基于索引实现的, 如果查询语句未命中任何索引, 那么 InnoDB 会使用表级锁。此外, InnoDB 行级锁是针对索引加的锁, 不针对数据记录, 因此即使访问不同行的记录, 如果使用到了相同的索引键仍然会出现锁冲突。还需要注意的是, 在通过 `SELECT ... LOCK IN SHARE MODE;` 或者 `SELECT ... FOR UPDATE` 使用锁的时候, 如果表没有定义任何索引, 那么 InnoDB 会创建一个隐藏的聚簇索引并使用这个索引来加记录锁。

此外, 不同于 MyISAM 总是一次性获得所需的全部锁, InnoDB 的锁是逐步获得的。当两个事务都需要获得对方持有的锁, 导致双方都在等待, 这时就产生了死锁。发生死锁后, InnoDB 一般都可以检测到, 并使一个事务释放锁回退, 另一个则可以获取锁完成事务。我们可以采取以下的方式避免死锁。

- 通过表级锁来减少死锁产生的概率。
- 多个程序尽量约定以相同的顺序访问表 (这也是解决并发理论哲学家就餐问题的一种思路)。
- 同一个事务尽可能做到一次锁定所需要的所有资源。

## 事务隔离案例

从某一账户转账  $n$  元给另一个账户, 数据库执行流程如下。

- 先从数据库读取自己的账户金额:

```
select amount from user_account where id = 1;
```

- 判断余额是否大于  $n$ , 如果小于  $n$ , 则结束事务; 大于  $n$ , 则更新余额记录:

```
update user_account set amount=amount - n where id = 1;
```

- 向对方账户增加  $n$  元:

```
update user_account set amont = amount + n where id = xx;
```

注意第一步, 如果是上面这种使用方式, 那么除 `Serializable` 隔离级别中同时并发的两个事务, 都可能会读取到相同的余额, 而后面不管怎么做都会使整体逻辑是错误的。可以使用 `Serializable` 来解决, 但是 `Serializable` 隔离级别一开始会给查询加共享锁, 并发事务也会同时用有相同记录的共享锁, 从而造成两个事务形成死锁, 都不能进行后续的 `update` 操作。

比较好的方式就是在第一步使用:

```
select amount from user_account where id = 1 for update;
```

此语句会直接给目标记录加排他锁，这样就防止出现并发事务读取到同样的余额数据造成业务错误。

当然，在业务应用层面做并发控制是另一种思路。

### 5.1.7 大表优化

当 MySQL 单表记录数过大时，数据库的 CRUD 性能会下降，需要如下的一些优化措施。

#### 1) 限定数据的范围。

对于大数据量的数据表，全表扫描肯定是不可接受的。务必禁止不带任何限制数据范围条件的查询语句存在。比如，当用户查询订单历史数据时，可以控制在最近一个月的数据中进行筛选。

#### 2) 读 / 写分离。

经典的数据库拆分方案主库负责写，从库负责读。

#### 3) 缓存。

- 使用 MySQL 的查询缓存。
- 对重量级、更新少的数据考虑使用应用级别的缓存。

#### 4) 垂直分区。

根据数据库里面数据表的相关性进行拆分。例如，用户表中既有用户的登录信息又有基本信息，可以拆为两个单独的表做分表，甚至放到单独的库做分库。垂直分区的优点在于可以使得行数据变小，在查询时减少读取的 Block 数，减少 I/O 次数。此外，垂直分区可以简化表的结构，易于维护。

垂直分区的缺点在于主键会出现冗余，需要管理冗余列，并会引起 Join 操作，可以通过在应用层进行 Join 来解决。此外，垂直分区会让事务管理变得复杂。

#### 5) 水平分区。

水平分区指的是保持数据表结构不变，通过某种策略存储数据分片。这样每片数据分散到不同的表或者库中，达到了分布式的目的。水平分区可以支撑非常大的数据量。

如果某个表的数据是时间序列的，比如订单、交易记录等，通常比较适合用时间范围分片，因为具有时效性的数据，我们往往更关注其近期的数据，查询条件中一般带有时间字段可以进行过滤。可以使用跨度短的时间范围分片活跃数据，跨度长的时间范围分片历史数据。此外，订单表包含正在处理和已完成的订单，而正在处理的订单是频繁被访问查



询的，已完成订单则相对来说很少被访问，那么订单状态也可以作为分片的因子，这样可使得频繁访问的正在处理的订单能够保证一个相对小的规模，从而提高处理速度。这里分离频繁和不频繁使用的数据也是大表优化的一个原则。

需要注意的是，分表仅仅解决了单一表数据过大的问题，但由于表的数据还是在同一机器上，其实对于提升 MySQL 并发能力没有什么意义。所以水平分区最好分库。

水平分区能够支持非常大的数据量存储，应用端的改造也较少，但分片事务难以解决，跨节点 Join 性能差，逻辑复杂。

MySQL 5.1 之后提供的分区表也是水平分区，用户需要在建表的时候加上分区参数，这对应用是透明的，无须修改代码：

```
CREATE TABLE `user_note` (
  id BIGINT(20) NOT NULL,
  uid BIGINT(20) NOT NULL,
  content varchar(1024) NOT NULL,
  create_time DATE NOT NULL
)
PARTITION BY RANGE( YEAR(create_time) ) (
  PARTITION p0 VALUES LESS THAN (2013),
  PARTITION p1 VALUES LESS THAN (2016),
  PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

除此之外，数据库分片主要包括两种分片方案，分别是客户端代理和中间件代理。

- 客户端代理

分片逻辑在应用端，封装在 jar 包中，通过修改或者封装 JDBC 层来实现。当当网的 Sharding-JDBC、阿里的 TDDL 是目前比较为人所知的实现。

- 中间件代理

在应用和数据中间加了一个代理层。分片的逻辑统一维护在中间件服务中。阿里的 Cobar、360 的 Atlas、网易的 DDB、开源的 MyCat 和 Kingshard 都是这种架构的实现。

笔者推荐如非特别必要，不要对数据进行分片，拆分会带来逻辑、部署、运维的各种复杂度，一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的。如果实在要分片，尽量选择客户端分片架构，毕竟减少了一次和中间件的网络 I/O。

## 5.1.8 高可用支持

MySQL 默认支持主从配置，可以一主一从，也可以一主多从。但是这个主从仅仅实现了读/写分离，并不能解决高可用的问题。常用的高可用方案如下。

## 1) MHA

MHA, Master High Availability, 是目前 MySQL 中相对成熟的高可用解决方案, 在互联网公司中被经常使用。此种方案可以保障一主多从的高可用, 管理节点会定时探测集群中的 Master 节点, 当 Master 出现故障时, 自动将最新数据的 Slave 提升为新的 Master, 然后将所有其他的 Slave 重新指向新的 Master。整个故障转移过程对应用程序完全透明。架构如图 5-2 所示。

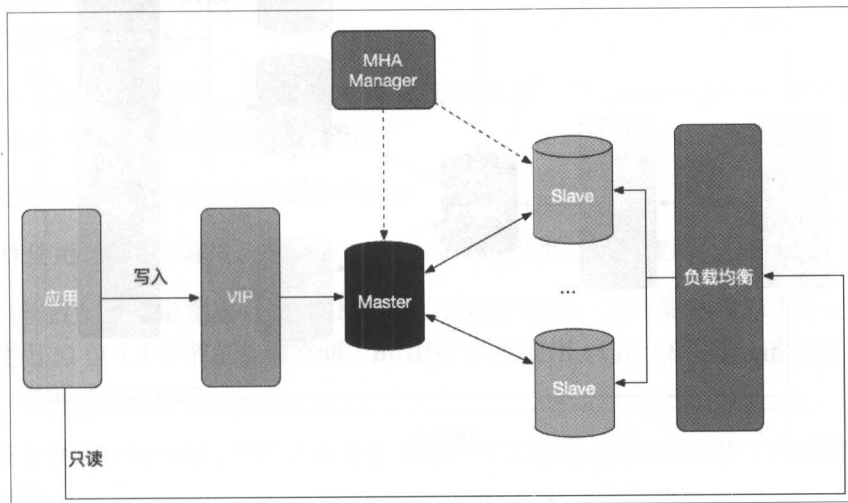


图 5-2

## 2) MMM

MMM, Multi-Master Replication Manager, 是双主的架构方案, 同时只有一个主允许写, 另一个主允许读, 一个主挂掉, 其下面的 Slave 同样挂掉。此方案无法严格保证数据一致性, 适用于对数据一致性要求不高的业务场景。架构如图 5-3 所示。

前面水平分区中介绍的中间件代理基本也都是采用这两种方案或者类似方案做的高可用保证。

除了上述两种方案, 还有双主配 SAN 存储、双主配 DRBD、NDB CLUSTER、镜像等高可用方案由于成本、复杂性等原因并未被广泛应用。

此外, 在主从模式下, 经常遇到的一个问题是 Slave 数据滞后于 Master, 写入 Master 后从 Slave 读取不到最新的数据。可以使用 MySQL 5.5 及之后版本引入的半同步复制机制, 使得在 MySQL 客户端请求时阻塞, 一直到数据至少已经同步到一个 Slave 或者超时, 如此可以在一定程度上解决数据一致性的问题。

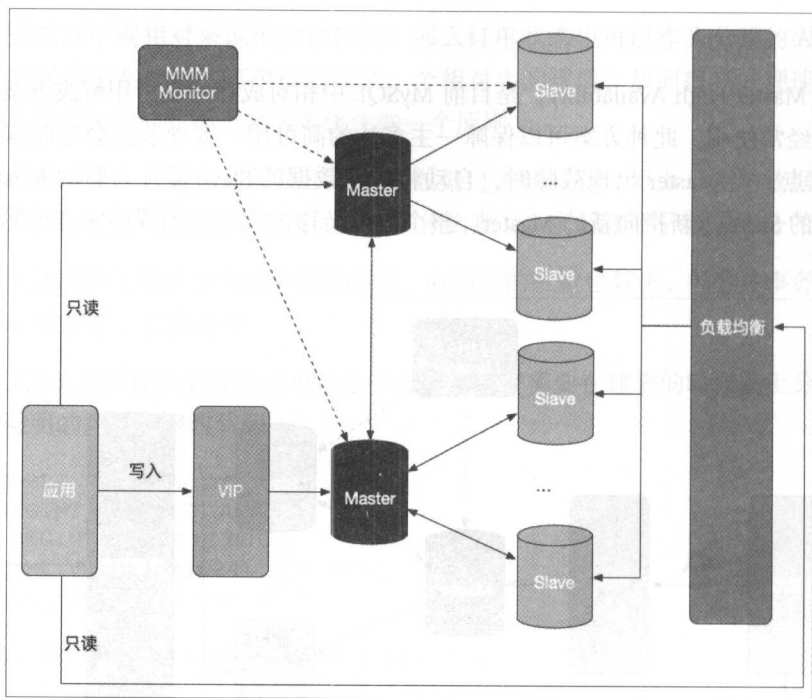


图 5-3

### 5.1.9 使用提示

#### MySQL 使用

MySQL 本身的使用和配置有以下注意点。

- MySQL 默认的连接数是 100，调整 MySQL 的最大连接数能够在一定程度上提高并发能力。
- 使用慢查询日志监测导致数据库响应缓慢的 SQL 语句。
- MySQL 有一个常见的连接缓慢原因，那就是会根据域名做反向 DNS 查询。这个问题可以通过在 MySQL 中加入 skip-name-resolve 选项来解决（同时需要在 MySQL 用户权限中把 root@127.0.0.1 都配置上），还可以通过在 hosts 中加入本机域名的映射来解决。
- 在 MySQL 命令行中命令以 \G 结束，改变结果显示方式为列。
- 通过提升 CPU 和内存、使用 SSD，能显著提升 MySQL 性能。

- 可以使用 PT 工具箱 (Percona Toolkit) 对 MySQL 进行管理, 包括检查主从复制的数据一致性、检查重复索引、在线 DDL、定位 I/O 占用高的表文件等。

## DDL

在设计数据库的时候有以下注意点和技巧。

- 禁用存储过程、函数、触发器、外键约束, 尽量依赖于业务层面做, 这样能够具有良好的可扩展性。
- 允许为 NULL 的列, 在非查询时会无法匹配, 如 `where status != 'FINISH'`, 那么 `status` 为 NULL 的行也无法匹配。
- 使用枚举或整数类型代替字符串类型。
- ID 使用 BIGINT 即可, 对应于 Java 中的 long 类型, 足够使用。
- 对整数类型指定宽度, 比如 INT(11)、BIGINT(20), 并不影响存储大小, INT 依然使用 32 位 (4 字节) 存储空间, BIGINT 依然使用 64 位 (8 字节) 存储空间, 宽度仅表示显示的长度。
- 避免使用 DECIMAL 和浮点数数据类型, 可以使用 BIGINT 代替 (浮点数乘以一个倍数)。
- 数据库中的表最好带有创建和更新时间戳, 以及所创建 / 修改行的用户标识, 以审计追踪数据的变动。
- 不要真的删除数据, 可以给它们打上一个被删除的标记或者做版本化修改。
- 单表不要有太多字段, 建议在 15 以内。
- 用整数类型而不是字符串存储 IP。

## DML

在数据查询的时候有以下注意点和技巧。

- 在查询语句中不要使用全属性选择器 `*`。
- 尽量避免使用负向查询 (`!` 和 `<>`) 和 `or` 查询, 而使用 `in` 代替。`in` 的查询效率是  $\log(n)$ 。
- 对于连续条件的查询, 使用 `between` 而不是 `in`。
- 模糊匹配查询时, 当为后缀模糊查询时能够使用索引, 如 `like 'a%'`, 而前缀则不行。

- 查询记录数目时，使用 `count(1)` 和 `count(*)` 是等价的，在有索引的时候都会去选择合适的索引，没有索引的时候则全表扫描。而 `count(column)` 则是统计 `column` 不为 `NULL` 的记录数目。
- 避免多于两表 Join，尽量使用冗余策略解决联表问题。
- 避免列运算，如 `select * where age + 1 > 100`。
- 批量插入代替循环单条插入，能够减少网络 I/O 带来的开销。
- 一个很耗时的 SQL 会堵死整个库，可以拆开多个小的语句进行，以减少锁的时间。
- 使用 UNION 时，除非确实需要服务器去重，否则一定要使用 UNION ALL。若不是这样，MySQL 会给临时表加上 DISTINCT 选项，导致整个临时表的数据做唯一性检查，影响查询性能。
- 插入一条记录后，如果这张表的主键是自增的，推荐使用 `SELECT LAST_INSERT_ID()` 来获取这个自增值，`LAST_INSERT_ID` 是基于 Connection 的，只要 Connection 对象不变，获取到的自增 ID 就是正确的，也不需要加锁。
- 若某一数据列为 bit 类型，那么在用 MySQL 命令行查询的时候是无法看到其值的。需要如下查询：

```
select bin(bit_column + 0) from test_user; // 显示二进制数
```

```
select bit_column + 0 from test_user; // 显示十进制数
```

更多的 MySQL 常用命令可以查看附录 C。

## 数据库连接池

在 Java 应用中使用关系型数据库时，会使用数据库连接池以避免频繁创建、销毁连接带来的性能开销。市面上有很多数据库连接池，主流的有下面几个。

- HikariCP：是 BoneCP 的作者推出的 BoneCP 的替代品，号称有了质的变化、革命性的变更。
- Druid：阿里巴巴开源的数据库连接池，还提供了一些配套监控工具、统计功能和 Web 界面，性能也较好。
- DBCP：老牌的数据库连接池，现在到了 DBCP2，是基于 commons-pool 之上的封装。
- C3P0：是与 Hibernate 一起发布的开源数据库连接池。

以上数据库连接池综合对比如表 5-2 所示。

表 5-2

	是否支持 PSCache	监控	扩展性	SQL 拦截及解析	代码	特点
HikariCP	否	JMX	弱	无	简单	功能简单，性能好，起源于 BoneCP
Druid	是	JMX/Log/HTTP	好	支持	中等	功能全面，方便对 JDBC 接口进行监控追踪
DBCP	是	JMX	弱	无	简单	依赖于 commons-pool
C3P0	是	JMX/Log	弱	无	复杂	历史悠久，代码逻辑复杂，且不易维护

表 5-2 中的 PSCache 指的 preparedStatement 缓存，是 connection 私有的，key 为 prepare 执行的 SQL 和 Catalog 等，value 对应的为 preparedStatement，支持此特性可以减少解析 SQL 的开销，对性能会有大概 20% 的提升。

综上，如果对监控、SQL 统计有需求，推荐使用 Druid；如果特别关注性能，那么推荐使用 HikariCP。

5.2 非关系型数据库

5.1 节介绍了关系型数据库，事实上很多业务中的数据表并不要求 ACID 和事务，可以考虑使用 NoSQL 数据库，这样能够彻底解决水平扩展问题。

非关系型数据库是相对关系型数据库来讲的，又被称为 NoSQL 数据库，也可以叫作 Not Only SQL 数据库。相比传统的 SQL 关系型数据库，其最大的特点就是适合存储非结构化或半结构化的数据，适合存储大规模数据。以键值对存储，且结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对，这样就不会局限于固定的结构，可以减少一些时间和空间的开销。但是，其没有关系型数据库那种严格的数据模式，并不适合复杂的查询以及需要强事务管理的业务。

常用的 NoSQL 数据库主要有以下几种。

- **KV 数据库**：主要以 (key,value) 键值对存储数据的数据库。以 Redis、SSDB 为代表。
- **文档数据库**：总体形式上也是键值对的形式，但是值又可以有各种数据结构：数组、键值对、字符串等。以 MongoDB、CouchDB 为代表。
- **稀疏大数据库（列数据库）**：一般是用来存储海量数据的。相对于行数据库，这种数据库是以列为单位在介质上存储数据的。以 HBase、Cassandra 为代表。

还有诸如图数据库、对象数据库、XML 数据库等都属于 NoSQL 数据库。

此外，这里需要提到 CAP 这个理论，它的核心在于一个分布式系统不可能同时满足以下 3 点，最多能够较好地同时满足两个。

- **Consistency**: 一致性，所有节点在同一时间具有相同的数据。
- **Available**: 可用性，每次请求都能在期望时间内获取正确的响应，但不保证获取的数据为最新数据，可以允许数据不一致。
- **Partition tolerance**: 分区容错性，系统中任意信息的丢失或失败不会影响系统的继续运作。

根据 CAP 理论，可以将数据库分为如下 3 类。

- **CA**: 传统关系型数据库的单点模式，满足数据的一致性和高可用性，但没有可扩展性。
- **CP**: MongoDB、HBase，满足数据的一致性和分区性，通常性能不是特别高。
- **AP**: Cassandra，具有较好的性能和可扩展性，但各节点之间的数据同步比较慢，能保证数据的最终一致性。

此外，基于 CAP 理论，为了获得可扩展性和高可用性，BASE 原则提出了对 NoSQL 数据库的可用性及一致性的弱要求。如下：

- 1) **Basically Available**: 基本可用，允许系统暂时不一致。
- 2) **Soft state**: 软状态 / 柔性事务，可以理解为“无连接”的，暂时允许一些不准确的地方和数据的变换。
- 3) **Eventual Consistency**: 最终一致性，当所有服务逻辑执行完成后，系统最后将回到一个一致的状态。

可以看出，和 ACID 关注数据完整性和一致性相比，BASE 的首要目标是可用性。保证在短时间内，即使有不同步的风险，也要允许新数据能够被存储。

## 5.2.1 KV 数据库

KV 数据库是主要以 (key,value) 键值对存储数据的数据库，可以通过 key 快速查询到其 value，包括 Redis 和 SSDB 等。

### 1) Redis

Redis 开始是作为缓存使用的，但是由于其具有持久化的特性，因此很多时候被用作数据库。由于其主要的特性是缓存，因此本节不做详述。

## 2) SSDB

对于 SSDB, 需要先提到 LevelDB。LevelDB 是来自 Google 的一个快速、轻量级的 KV 存储类库, 能够实现持久化的存储。它是一个嵌入式数据库, 没有分布式支持、没有 schema, 也不支持事务, 仅仅是一些有序的 key 到 value 的映射, 在使用 SSD 硬盘作为存储介质的情况下能发挥最大优势。其使用了 LSM (Log Structure Merge) 树作为底层数据存储结构, 每次更新会记录提交 Log, 并提交到 in-memory table 中。之后 Memory 的数据落地是延迟的, 并采用一定的策略定期将磁盘中的数据进行 compact, 是针对写进行优化的一种存储设计。

SSDB 是 LevelDB 的 Redis 兼容协议封装 (使用 LevelDB 作为存储引擎), 并且实现了主从同步和持久化的队列服务, 支持 Redis 客户端。目前支持 key-string、key-set 以及 key-hashmap 这 3 种数据结构, 可以作为 Redis 之外的另一种选择。由于其是基于文件存储系统的, 因此其支持的容量可以很大, 但也受限于文件存储, 性能上相比 Redis 还是有一定的差距的, 且由于要进行 compact, CPU 的使用率有时候会很高。如果使用 SSD 硬盘作为存储设备, 性能则会更加接近 Redis。

此外, 还需要提到的是 RocksDB, 这个数据库存储引擎来自 Facebook, 是基于 LevelDB 存储引擎构建的一个可嵌入式的支持持久化的 key-value 存储系统, 也可作为 C/S 模式下的存储数据库, 但其主要目的还是嵌入式。其主要针对闪存做了低延迟优化, 并提供了随 CPU 数目进行线性扩展的能力。

## 5.2.2 文档数据库——MongoDB

文档数据库总体形式上也是键值对的形式, 但是值又可以有各种数据结构: 数组、键值对、字符串等, 可以认为是 JSON 格式的数据存储, 因此能够对任意字段建立索引, 实现关系型数据库的某些功能。目前 Java 中最常用的文档数据库是 MongoDB, 因此这里主要针对 MongoDB 讲述, MongoDB 为 3.2.7 版本。

MongoDB 是一个分布式文件存储的数据库, 是非关系型数据库中最像关系型数据库的。其具有以下特点。

- 灵活模式: 数据以 JSON 格式存储。
- 高可用性: 复制集保证高可用。
- 可扩展性: 通过 Sharded cluster 保证可扩展性。



## 关键概念

### 1) 数据库 (Database)。

对应于 MySQL 的 Database。

### 2) 集合 (Collection)。

对应于 MySQL 的 Table, 存在于数据库中, 没有固定的结构, 可以插入不同格式和类型的数据。

### 3) 文档 (Document)。

对应于 MySQL 的 Row, 是一组键值对 (BSON)。不需要设置相同的字段, 并且相同的字段不需要相同的数据类型。集合由文档构成。

### 4) 域 (Field)。

对应于 MySQL 的 Column。

### 5) 主键 (Primary Key)。

和 MySQL 相同, MongoDB 会自动将 `_id` 字段设置为主键。

### 6) 复制集 (Replica Set)。

MongoDB 复制集由一组 `mongod` 实例 (进程) 组成, 包含一个 Primary 节点、多个 Secondary 节点以及可选的 Arbiter 节点 (用于仲裁)。

MongoDB Driver (客户端) 的所有数据都写入 Primary, Secondary 从 Primary 同步写入的数据, 以保持复制集内所有成员存储相同的数据集, 提供数据的高可用。这类似于 MySQL 中的一主多从配置。

复制集保证了 MongoDB 的高可用性。

### 7) 分片集 (Sharded cluster)。

即 MongoDB 的分布式特性, 通过分片来构成分布式集群。由如下 3 个组件构成。

- **shard:** 数据节点, 每一个数据节点存储一部分分片数据, 每个 shard 都可以做复制集。
- **mongos:** 路由节点, 作为客户端和分片集群间的查询路由。
- **config server:** 配置节点, 存储了集群的元数据和配置信息。

目前主要支持如下两种数据分布策略。

- 范围分片 (Range based sharding) : 能很好地满足范围查询的需求, 但如果 shard key 有明显递增 (或者递减) 趋势, 则新插入的文档多会分布到同一个 Chunk, 无法扩展写的能力。
- Hash 分片 (Hash based sharding) : 根据用户的 shard key 计算 hash 值 (64 位整型数据), 根据 hash 值按照范围分片的策略将文档分布到不同的 Chunk。其与范围分片互补, 能将文档随机地分散到各个 Chunk, 充分扩展写的能力, 弥补了范围分片的不足, 但其不能高效地服务范围查询, 所有的范围查询都要分发到后端所有的 Shard 才能找出满足条件的文档。

Sharded cluster 保证了 MongoDB 的可扩展性。

## 存储引擎

一开始 MongoDB 仅仅支持 MMAP 存储引擎, 到了目前的 MongoDB 3.2, 已经支持多存储引擎。

### 1) MMAPv1

这是 MongoDB 最开始的存储引擎, 也是 3.2 版本之前的默认存储引擎。这就是比较简单的内存映射文件, 支持集合级别的并发控制, 不支持压缩。

### 2) WiredTiger

这是 MongoDB 3.0 引入的新的存储引擎, 支持 MongoDB 的所有特性, 是 3.2 版本的默认存储引擎。其提供文档级别 (Document-Level) 的并发控制、检查点 (CheckPoint)、数据压缩和本地数据加密 (Native Encryption) 等功能。

### 3) In-Memory

In-Memory 存储引擎用于将数据只存储在内存中, 只将少量的元数据和诊断日志 (Diagnostic) 存储到硬盘文件中, 由于不需要 Disk 的 I/O 操作就能获取所需的数据, In-Memory 存储引擎大幅度降低了数据查询的延迟 (Latency), 其支持文档级别的并发控制。此存储引擎是 MongoDB 企业版本的特性, 且只能用在 MongoDB 64 位版本上。

综上, 如果你使用的版本支持 WiredTiger, 那就使用 WiredTiger。

可以通过 mongod 参数指定存储引擎:

```
mongod --storageEngine wiredTiger | inMemory
```

## 索引

MongoDB 的索引数据结构也是 B 树，和之前介绍过的 MySQL 索引基本一致。不过这里需要注意的是，由于其底层采用的是内存映射文件，因此其是一种非聚集索引，尽管 MongoDB 本身并没有这种定义。

MongoDB 也提供了 explain 工具来查看查询的执行计划，如图 5-4 所示。

```

udb-xy1kw:PRIMARY> db.bbc_bb_host_message.find({groundId:10806}).sort({timeTag:1}).explain()
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "ssy_bbc.bbc_bb_host_message",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "groundId" : {
        "$eq" : 10806
      }
    },
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "groundId" : 1,
          "timeTag" : -1
        },
        "indexName" : "groundId_1_timeTag_-1",
        "isMultiKey" : false,
        "isUnique" : false,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 1,
        "direction" : "backward",
        "indexBounds" : {
          "groundId" : [
            "[10806.0, 10806.0]"
          ],
          "timeTag" : [
            "[MinKey, MaxKey]"
          ]
        }
      }
    },
    "rejectedPlans" : [ ]
  }
}

```

图 5-4

其中的 winningPlan 就是将要使用的执行计划。这里和索引相关的字段如下。

- indexName: 出现此字段则表明命中了索引。
- indexBounds: 描述了使用索引的情况，给出了索引的遍历范围。

MongoDB 从 2.4 版本开始支持全文索引：

```

db.user_note.createIndex({title:"text"})

db.user_note.find({$title:{$search:"basketball"}})

```

## TTL

MongoDB 2.2 版本后引入了 TTL 集合，即支持对集合进行失效时间设置。当超过指定时间后，集合会自动清除超时的文档，这在保存一些诸如 Session 会话信息和存储缓存数据的时候非常有用。

以下代码即表示以 `create_time` 为标准，300 秒后删除相应的文档：

```
db.user_login_log.ensureIndex({"create_time": 1},{expireAfterSeconds:
300})
```

## 固定集合

MongoDB 中的 Capped Collections 是具有固定大小的集合，其吞吐性能较出色。固定集合其实就类似于一个环，当空间用完，那么就从头开始存储。

下面则创建了一个固定集合，其最大占用空间为 10000B，最大存储文档数目为 1000。  
size 优先级高于 max：

```
db.createCollection("cappedLogCollection",{capped:true,size:10000,max:1000})
```

可以将普通集合转换为一个固定集合：

```
db.runCommand({"convertToCapped":"user_note",size:10000})
```

固定集合和普通集合的一个区别是，其文档是按照插入顺序存储的，默认情况下的查询也是按照插入顺序返回的，可以通过 `$natural` 调整顺序。此外，固定集合是不允许删除文档的，只能通过 `drop` 删除所有文档。

下面即以插入顺序返回文档：

```
db.cappedCollection.find().sort( { $natural: -1 } )
```

固定文档一般用于存储日志信息、缓存等。

## 自增 ID

MongoDB 没有类似 MySQL 一样的自增 ID 功能。其 `_id` 默认就是 `ObjectId`，是自动生成的 12 字节唯一标识。如果想要使用自增 ID，那么需要通过编程实现。思路如下：

1) 创建一个计数器集合，如 `mongo_ids`。其具有两列：`collectionName` 和 `seqId`。

2) 使用 `query:{ collectionName: 'user_note'},update: {$inc:{seqId:1}}` 进行 `db.mongo_ids.findAndModify` 获取自增 ID。

```
db.mongo_ids.findAndModify(
{
  query:{collectionName: 'user_note' },
  update: {$inc:{seqId:1}},
  new:true
});
```

## 聚合操作

MongoDB 中的聚合使用 `aggregate()`，用于处理数据（诸如统计平均值、求和等），并返回计算后的数据结果。类似 SQL 语句中的 `count(*)`。

`user_note` 集合如下：

```
{
  _id: xxxx,
  createUser: xxx,
  status: xx
  ...
}
```

下面即可计算每个用户的笔记数目：

```
db.user_note.aggregate
(
  [
    { $group :
      {
        _id : "$createUser",
        num_tutorial : { $sum : 1 }
      }
    }
  ]
)
```

除了 `sum`，MongoDB 还支持 `avg`、`min`、`max` 等聚合表达式。

使用 `aggregate()` 进行聚合操作，查询速度较快。但 MongoDB 不允许单个聚合操作占用过多的系统内存（阈值为 20%），如果超过限制，则会直接终止操作。此外，聚合操作返回的结果集必须小于 16MB。

辅助 `aggregate()` 的还有管道，即将上一文档的处理输出到下一个管道。常用的管道操作符如下。

- `$project`：修改输入文档的结构。可以用来重命名、增加或删除域，也可以用于创建计算结果以及嵌套文档。
- `$match`：用于过滤数据，只输出符合条件的文档。`$match` 使用 MongoDB 的标准查询操作。

- \$limit: 用来限制 MongoDB 聚合管道返回的文档数。
- \$skip: 在聚合管道中跳过指定数量的文档, 并返回余下的文档。
- \$sort: 将输入文档排序后输出。
- \$group: 将集合中的文档分组, 可用于统计结果。

除了普通的聚合外, MongoDB 还提供了 MapReduce 来做聚合操作。MapReduce 简单地说就是将大批量的工作(数据)分解(Map)执行, 然后将结果合并成最终结果(Reduce)。能够在多台 Server 上并行执行, 可以非常灵活地计算复杂的聚合逻辑。但是其非常慢, 适用于离线大规模数据分析。

以下代码即可查询每个用户有多少状态为 normal 的记事:

```
db.user_note.mapReduce(
  function() { emit(this.createUser,1); },
  function(key, values) {return Array.sum(values)},
  {
    query:{status:"normal"},
    out:"note_total"
  }
).find()
```

其中,

- Map 函数必须调用 emit(key, value) 返回键值对。
- Reduce 函数将 key-values 变成 key-value, 也就是把 values 数组变成一个单一的值 value。

## 安全写入

上面提到 MongoDB 是 CP 特性, 牺牲了可用性。但其实通过安全写入机制, 配置合适的 WriteConcern 是可以得到一定的可用性的。

MongoDB 中一个写入操作的流程如下。

- 1) 数据首先写入到 MongoDB 缓存中。
- 2) 缓存定时异步刷写到 Journal 日志文件中, 这里 Journal 是 MongoDB 的预写日志, 相当于 MySQL 的 redo log。
- 3) Journal 定时异步刷写到 MongoDB 的数据文件中。
- 4) 如果插入操作或更新操作中包含索引列, 同时会维护索引结构。

5) 如果是副本集, 数据还会写到 oplog 定容集合中, 这里的 oplog 相当于 MySQL 的 Binlog, 用于从节点同步数据。

WriteConcern 这个参数描述了 MongoDB 在返回一个写操作成功前应该提供的保证, 即在执行到何种程度的时候, MongoDB 可以认为写操作成功并返回响应给客户端。越严格的安全写级别, 越意味着要在执行更多步骤后才返回客户端响应, 也就意味着更长的响应时间。安全写级别主要包括两个选项: w 和 j, 其中 w 指返回前需要确认的次数 (比如 w=1 表示只需要主节点确认, w=2 表示主节点和至少一个从节点确认, j=1 表示需要成功写到 journal 文件)。

一般来说, 使用默认的设置 w=1、j=0 即可, 即主节点确认收到写请求即返回。这样就牺牲一定的数据一致性来得到一定的可用性。

因此, 要确保写入正确, 至少使用 WriterConcern.ACKNOWLEDGED; 对于不重要的数据, 则可以使 WriteConcern.UNACKNOWLEDGED 省去等待网络的时间。

## 高可用

MongoDB 支持主从模式, 在主服务器上开启 mongod 进程时加入参数 -master, 在服务器上开启 mongod 进程时加入 -slave 和 -source 指定主服务器, 这样在主数据库更新时, 数据被复制到从数据库中。下面以在单台机器开启两个节点做主从为例:

```
$ cd /data && mkdir mongodata_master mongodata_slave

$ ./mongodb/bin/mongod --port 27017 --dbpath /data/mongodata_master
--master &

$ ./mongodb/bin/mongod --port 27018 --dbpath /data/mongodata_slave
--slave --source localhost:27017 &
```

此外, MongoDB 还支持主主配置, 可以实现数据的双向同步, 但是在大部分情况下官方不推荐使用。主主模式下, 两个节点都可以写数据:

```
$ cd /data && mkdir mongodata_27050 mongodata_27051

$ ./mongodb/bin/mongod --port 27050 --dbpath /data/mongodata_27050
--master --slave --source localhost:27051 > /dev/null &

$ ./mongodb/bin/mongod --port 27051 --dbpath /data/mongodata_27051
--master --slave --source localhost:27050 > /dev/null &
```

这里需要注意的是, 主从模式只是一种简单的高可用部署, 当主节点或者一个从节点挂掉时, 仍然需要人工干预才能恢复系统的运行状态。与之相比, 官方更加推荐使用复制集的集群模式。

复制集与主从相比,多了心跳监测,当主节点挂点,会在集群内发起主节点的选举机制,自动选举一位新的主服务器。使用也比较简单,只需要把 `-master` 和 `-slave`、`-source` 参数去掉换成 `-replSet` 参数,并指定复制集名称。如下:

```
./mongodb/bin/mongod --port 27017 --dbpath /data/mongodata_master
--replSet testReplset
```

## GridFS

GridFS 是 MongoDB 中的一个内置功能,用于存储和获取超过 16MB (BSON 文件限制) 的文件,将其存储在集合中,是文件存储的一种。

GridFS 会将文件分割为小的文件片段(默认大小为 255KB,最后一个片段除外),每一个片段存储为一个单独的文档。

GridFS 使用两个集合存储一个文件: `fs.file` 用来存储文件元信息(filename、content\_type, 以及用户自定义的属性), `fs.chunks` 用来存储文件存储片段。GridFS 同时也会给这两个集合创建索引以提高读/写性能:

```
./mongofiles -d gridfs put test.dat

//fs.files
{
  "filename": "test.dat",
  "chunkSize": NumberInt(211120),
  "uploadDate": ISODate("2017-05-01T11:32:33.557Z"),
  "md5": "7b762939321e146569b07f72c62cca4f",
  "length": NumberInt(646)
}

//fs.chunks
{
  "files_id": ObjectId("334a7asdad19f54bfec8a2fe44b"),
  "n": NumberInt(0),
  "data": "Mongo Binary Data"
}
```

此外,在某些场景下,把文件存储在 GridFS 中要比直接存储在系统的文件系统里性能要更好,如下:

- GridFS 对目录下的文件数目没有限制,可以避开一些文件系统的限制,存放大量小文件。
- GridFS 可以仅仅读取文件的部分信息,这样当读取大文件时就不需要把这个文件都加载到内存中。
- GridFS 可以自动对文件做分布式管理。



## 使用提示

对 MongoDB 本身的使用需要注意以下几点。

- 建议开启 Journaling 功能，特别是对于可用性要求较高的用户。
- MongoDB 只支持单一文档的原子性。
- MongoDB 默认情况下是没有认证功能的，可以采取防火墙的方式进行保护，也可以进行相关的安全认证配置。
- 生产环境开启 profiling 功能，以监测慢请求。
- MongoDB 对内存的要求比较高，生产环境中 MongoDB 所在的主机应该配置尽量大的内存。

使用 MongoDB 进行数据操作需要注意以下几点。

- MongoDB 默认情况下是区分大小写的。
- 确保输入正确的数据类型，输入错误数据类型不会出现提示。
- MongoDB 的更新在默认情况下会使用类似于传统数据库的 LIMIT 语句，即 LIMIT 1。如果想要一次更新许多文档，需要把 multi 设为 true。
- MongoDB 的翻页效率比较低。使用 find().skip().limit() 的方式，务必要使用索引，否则会导致全表扫描几百万数据几乎不可用。
- 不要使用负向查询 \$nin、\$not，这样不会命中索引。
- 查询条件不要使用算术运算符，如 \$mod，这样不会使用索引。
- MongoDB 中存储的文档必须有一个“\_id”键，不指定值时，默认是一个 ObjectId 对象。
- 和 MySQL 一样，要减少不必要的或重复的索引，以避免更新或插入时对索引的维护开销。
- 使用写入文档数组或者 Bulk Write 进行批量写，减少网络请求次数。
- 作为主键的 \_id 值应该避免使用随机值（MD5、uuid 等），而应该使用自增值（比如默认的 ObjectId）。

MongoDB 的常用命令可查看附录 D。

### 5.2.3 列数据库——HBase

与传统的行数据库相比,列数据库的特点就在于其是以列为存储单位的,以此方便存储结构化和半结构化数据并做数据压缩,其对针对某一列或者某几列的查询有非常大的 I/O 优势。HBase 是最常用的列数据库之一。

HBase 是一个分布式的、面向列的开源数据库,该技术来源于 Google 论文“Bigtable: 一个结构化数据的分布式存储系统”。BigTable 是基于 GFS 的, HBase 是基于 HDFS 的, 是 Hadoop 生态的核心组件之一。它在一个表里存储数据行, 一个数据行拥有一个可选择的键和任意数量的列。表是疏松存储的, 因此用户可以给行定义各种不同的列。主要用于需要随机访问、实时读/写大数据的场景。

这里需要注意的是, HBase 是面向 OLAP 的一种数据库, 受限于底层的数据结构, 如果不做二次优化, 一般不推荐用于面向用户的业务。

概括来看, HBase 适用于以下存储场景。

#### 1) 半结构化或非结构化数据。

对于数据结构字段不够确定或杂乱无章很难按一个概念去进行抽取的数据适合用 HBase。当业务发展需要增加存储如一个用户的 E-Mail、address 信息时, RDBMS 需要停机维护, 而 HBase 支持动态增加。

#### 2) 记录非常稀疏。

RDBMS 的行有多少列是固定的, 为 NULL 的列会浪费存储空间。HBase 为 NULL 的 Column 不会被存储, 这样既节省了空间又提高了读性能。

#### 3) 多版本数据。

根据 RowKey 和 Column key 定位到的 Value 可以有任意数量的版本值, 因此对于需要存储变动历史记录的数据, 用 HBase 就非常方便。

#### 4) 超大数据量。

当数据量越来越大, RDBMS 数据库无法支撑, 就出现读/写分离和各种分库分表策略, 会带来业务复杂度的增加、无法 Join 等问题。采用 HBase 只需要加机器即可, HBase 会自动水平切分扩展, 跟 Hadoop 的无缝集成保障了其数据可靠性 (HDFS) 和海量数据分析的高性能 (MapReduce)。

## 关键概念

### 1) RowKey

行主键，HBase 不支持条件查询和 Order by 等查询，读取记录只能按 RowKey（及其 range）或全表扫描，因此 RowKey 需要根据业务来设计以利用其存储排序特性（Table 按 RowKey 字典序排序如 1、10、100、11、2）提高性能。

### 2) Column Family

列族，在表创建时声明，每个 Column Family 为一个存储单元。

### 3) Column

列，HBase 的每一列都属于一个列族，以列族名为前缀，如列 user:name 和 user:city 属于 user 列族，note:title 和 note:type 属于 note 列族。

Column 可以动态新增，同一 Column Family 的 Column 会聚簇在一个存储单元上，并依 Column key 排序，因此设计时应将具有相同 I/O 特性的 Column 设计在一个 Column Family 上以提高性能。

### 4) Timestamp

HBase 通过 row 和 column 确定一份数据，这份数据的值可能有多个版本，不同版本的值按照时间倒序排列，即最新的数据排在最前面，查询时默认返回最新版本。Timestamp 默认为系统当前时间（精确到毫秒），也可以在写入数据时指定该值。

### 5) Value

每个值通过 4 个键唯一索引：tableName + RowKey + ColumnKey + Timestamp => value。

### 6) 存储类型

- TableName 是字符串。
- RowKey 和 ColumnName 是二进制值（Java 类型 byte[]）。
- Timestamp 是一个 64 位整数（Java 类型 long）。
- value 是一个字节数组（Java 类型 byte[]）。

HBase 的 HTable 存储结构如下：

```
SortedMap{
    RowKey, List()
    SortedMap(
        Column, List(
            Value, Timestamp
```

即 HTable 按 RowKey 自动排序，每个 Row 包含任意数量的 Column，Column 之间按 Column key 自动排序，每个 Column 包含任意数量的 Value。

## 关键实现

HBase 有几个本身架构设计的组件需要了解。

- **Zookeeper 群**：HBase 集群中不可缺少的重要部分，主要用于存储 Master 地址、协调 Master 和 RegionServer 等上下线、存储临时数据等。
- **Master 群**：Master 主要是做一些管理操作，如 Region 的分配、手动管理操作下发等，一般数据的读 / 写操作并不需要经过 Master 集群，所以 Master 不需要很高的配置即可。
- **RegionServer 群**：RegionServer 群是数据真正存储的地方，每个 RegionServer 由若干个 Region 组成，而一个 Region 维护了一定区间 RowKey 值的数据。简称 RS。
- **HLog**：是 HBase 实现 WAL 方式产生的日志信息，其内部是一个简单的顺序日志，每个 RS 上的 Region 都共享一个 HLog，所有对于该 RS 上的 Region 数据写入都被记录到该 HLog 中，以备在 RS 出现意外崩溃的时候，可以尽量多地恢复数据。
- **MemStore**：可以看作 HBase 的内部缓存，每次数据写入完成 HLog 后，都会写入对应 Region 的 MemStore。
- **HFile**：HBase 的数据底层存储。每次数据从 MemStore 中 flush，最终都会形成一个 HFile。

基于以上几个集群，先看一下如何定位 RegionServer，这里涉及两个特殊的表：Meta 和 Root，用于存储数据库的元信息。

- **Meta 表**中记录了 RowKey 是在哪个 Region 的范围以及各个 Region 是在哪个 RegionServer 上等待等信息，是查询 HBase 时首先要访问的表。其 RowKey 设计为：Region 所在的表名 + Region 的 StartKey + 时间戳，三者的 MD5 值是 HBase 在 HDFS 上存储的 Region 的名字。此外，当 Region 被拆分、合并或者重新分配的时候，都需要修改这张表的内容。
- **Root 表**：记录了 Meta 表的 Region 信息，Root 表只有一个 Region，其位置信息记录在 ZooKeeper 中

一个 Region 的定位过程如下。

- 1) 读取 ZooKeeper 中 Root 表的位置信息。
- 2) 通过 Root 表获取 Meta 表的位置。
- 3) 读取 .Meta 表中用户表的位置。
- 4) 读取数据。

如果已经读取过一次, 则 Root 表和 Meta 表都会缓存到本地, 直接去用户表的位置读取数据即可。

定位到 RS 之后, 就可以进行数据写入和读取。

数据的写入首先要写 HLog 以实现 WAL, 写完 HLog 之后, 按照 RowKey 的值排序写入 MemStore, 即成功返回。存储于 MemStore 中的数据是 LSM 的数据结构, 需要不定期地进行 compact 以减少文件碎片数、提高性能。这里需要注意的是, Hlog 在数据 flush 到磁盘后, 会被移动到 .oldlogs 这个目录下, 会有一个 HLog 监控线程监控该目录下的 HLog, 根据设置删除过期的 HLog, 以防 Hlog 浪费存储空间。此外, HBase 基于的文件系统 HDFS 是 append only 的, 因此数据的删除和过期一开始只是被标记为删除, 在 compact 时才真正删除。

另外需要注意的一点是, 在数据 flush 的时候, 对应 Region 上的访问都是被拒绝的, 因此控制 flush 的时机是非常重要的。主要有以下几种方式会触发 flush。

- 通过全局内存控制, 触发 MemStore 刷盘操作。通过参数 `hbase.regionserver.global.memstore.upperLimit` 进行设置, 内存下降到 `hbase.regionserver.global.memstore.lowerLimit` 配置的值后, 即停止 MemStore 的刷盘操作。
- HBase 提供 API 接口, 运行通过外部调用进行 MemStore 的刷盘。
- 前面提到 MemStore 的大小通过 `hbase.hregion.memstore.flush.size` 进行设置, 当 Region 中 MemStore 的数据量达到该值时, 会自动触发 MemStore 的刷盘操作。
- WAL 达到阈值, 会引起 MemStore 的 flush。WAL 的最大值由 `hbase.regionserver.maxlogs * hbase.regionserver.hlog.blocksize` (2GB by default) 决定。

还需要注意的是, 一个列族达到阈值触发 flush 的时候, 也会导致其他的列族 flush, 因此列族的数量越少越好。

相比数据的写入, 数据的读取相对来说比较简单: HBase 首先检查请求的数据是否在 MemStore, 不在的话就到 HFile 中查找, 最终返回 merged 的一个结果给用户。

## 使用提示

- RowKey 的设计越短越好，尽量不要超过 16 字节。
- 避免使用时序或者单调（递增 / 递减）行键，否则会导致连续到来的数据被分配到同一 Region 中，可以采取在 RowKey 前面添加 MD5 散列值的方式。
- 列族的数量越少越好，否则会造成在数据查询的时候读取更多的文件，消耗更多的 I/O。
- 同一个表中不同列族所存储的记录数量的差别（列族的势）会造成记录数量少的列族的数据分散在多个 Region 上，影响查询效率。
- 尽量最小化行键和列族的大小，避免 HBase 的索引过大，加重系统存储的负担。
- HColumnDescriptor 设置版本的数量，避免设置过大，版本保留过多。
- 列族可以通过设置 TTL 来实现过期失效。
- 为避免热点数据产生和后续文件 split 影响业务使用，一般采用 hash + partition 的方式预分配 Region，首先使用 MD5 hash，然后按照首字母 partition 为 32 份，就可以预分配 32 个 Region。
- Region 的数量选择可以参考此公式：一个 RS 的内存消耗 = MemStore 大小 \* Region 数量 \* 列簇数量。
- HBase 支持多种形式的数据压缩，如 GZip、LZO、Snappy 等。其中 Snappy 的压缩率最低，但是编解码速率最高，对 CPU 的消耗也最小，一般使用 Snappy 即可。
- 对于有随机读的业务，建议开启 Row 类型的过滤器，使用空间换时间，提高随机读性能。
- 避免全表扫描 HBase 数据。

## 5.3 缓存

缓存是为了弥补持久化存储服务，如数据库的性能缓慢，而出现的一种将数据存储在内存中，从而大大提高应用性能的服务。如缓存五分钟法则所讲：如果一个数据频繁被访问，那么就应该放内存中。这里的缓存就是一种读写效率都非常高的存储方案，能够应对高并发的访问请求，通常情况下也不需要持久化的保证。但相对其他存储来说，缓存一般是基于内存的，成本比较昂贵，因此不能滥用。

缓存可以分为本地缓存和分布式缓存。

### 5.3.1 本地缓存

本地缓存指的是内存中的缓存机制，适用于尺寸较小、高频的读取操作、变更操作较少的存储场景。在 Java 开发中常用的本地缓存实现如下。

#### 1) ConcurrentMap

这是 JDK 自带的线程安全 map 实现，适合用户全局缓存。其 get、put 的操作比较简单，不用赘述。如果想要实现缓存的失效、淘汰策略，则需要自定义实现。

#### 2) LinkedHashMap

LinkedHashMap 也是 JDK 的实现。其一个简单的用途是一个可以保持插入或者访问顺序的 HashMap，但其实若配置好是可以当作 LRU cache 的。这里的 LRU 即 Least Recently Used，指的是固定容量的缓存，当缓存满的时候，优先淘汰的是最近未被访问的数据：

```
int cacheSize = 1000; // 最大缓存 1000 个元素

LinkedHashMap cache = new LinkedHashMap<String, String>(16, 0.75f, true) {
    @Override
    protected boolean removeEldestEntry(Map.Entry<String, String> eldest)
    {
        return size() > cacheSize;
    }
};
```

需要注意的是，LinkedHashMap 是非线程安全的，如果是全局使用，需要做并发控制。

#### 3) Guava Cache

Guava Cache 来自于 Google 开源的 Guava 类库，是一个实现比较完全的本地缓存，包括缓存失效、LRU 都做了支持。

下面的 load 方法是第一次加载对应的 key 的缓存时调用的方法，重载此方法可以实现单一线程回源，而 reload 方法的重载，则可以在后台定时刷新数据的过程中依然使用旧数据响应请求，不会造成卡顿，这里默认的实现是 load 方法的代理，是同步的，建议重新用异步方式实现。此外，并行度指的是允许并行修改的线程数，此值建议根据当前机器的 CPU 核数来设置。

```
final int MAX_ENTRIES = 1000; // 最大元素数目
LoadingCache<String, String> cache = CacheBuilder.newBuilder()
    .maximumSize(MAX_ENTRIES)
    .concurrencyLevel(Runtime.getRuntime().availableProcessors()) // 并行度
    .expireAfterWrite(2, TimeUnit.SECONDS) // 写入 2 秒后失效
    .build(new CacheLoader<String, String>() {
        @Override
        public String load(String key) throws Exception {
```

```

        return ...; // 异步加载数据到缓存
    }

    @Override
    public ListenableFuture<String> reload(String key, String
oldValue) throws Exception {
        return ...;
    }
});

//Using the cache
String value= cache.getUnchecked("testKey");

```

上述例子中使用了基于 `maximumSize` 和基于时间 `expireAfterWrite` 的缓存剔除，除此之外，还可以通过：

### 1) 基于权重的缓存剔除。

```

CacheBuilder.newBuilder()
    .maximumWeight(10000)
    .weigher(new Weigher<String, Object>() {
        @Override
        public int weigh(String key, Object value) {
            return key.length();
        }
    })
    .build();

```

这样当 `cache` 中 `put` 一个 `key` 时，都会计算它的 `weight` 值并累加，当达到 `maximumWeight` 阈值时，会触发剔除操作。

### 2) 制定 `key` 和 `value` 使用的引用类型来做缓存剔除。

```

CacheBuilder.newBuilder().weakKeys();
CacheBuilder.newBuilder().weakValues();
CacheBuilder.newBuilder().softValues();

```

还需要指明的一点是，Guava Cache 中的缓存失效并非立即生效，通常是延迟的，在各种写入数据时都去检查并 `cleanUp`。

此外，Guava Cache 还提供了 `asMap` 视图，可以获取保存数据使用的 `ConcurrentMap` 形式。使用此视图时需要注意读 / 写操作会重置相关缓存项的访问时间，包括 `asMap().get()` 方法和 `Cache.asMap().put()` 方法，但 `asMap().containsKey()` 方法和遍历 `asMap().entrySet()` 除外。

这里还需要提到的一点是，缓存框架 Caffeine 使用 Java 8 对 Guava 进行了重写，包括驱逐策略、过期策略和并发机制，使得缓存性能得到了显著提升，并且使用上可以兼容 Guava 的 API。如果是在 Java 8 上的开发，推荐直接使用 Caffeine 作为本地缓存实现。



```

LoadingCache<String, String> cache = CaffeinatedGuava.build(
    Caffeine.newBuilder().maximumSize(MAX_ENTRIES),
    new CacheLoader<String, String>() { // Guava's CacheLoader
        @Override
        public String load(String key) throws Exception {
            return "";
        }
    });

```

### 5.3.2 分布式缓存——Redis

分布式缓存指的是单独的缓存服务，可独立部署，通过协议、接口等提供缓存服务。相比本地缓存，能够支持更大的容量。

几年前最流行的分布式缓存软件是 Memcached，但其支持的数据结构太少，现在已经基本被 Redis 取代。Redis 能够支持丰富的数据结构，基于事件驱动的单线程非阻塞 I/O 也能够应对高并发的业务场景。这里主要针对 Redis 来讲述，Redis 为 3.2.10 版本。

Redis 的功能非常强大，既可以作为数据库也可以作为缓存，还能当作队列。总体概括来讲，其有以下用途。

- 最简单的 String，可以作为 Memcached 的替代品，用作缓存系统。
- 使用 SetNx 可以实现简单的分布式锁。
- 使用 list 的 Pop 和 Push 功能可以作为阻塞队列 / 非阻塞队列。
- 使用 SUBSCRIBE 和 PUBLISH 可以实现发布 / 订阅模型。
- 对数据进行实时分析，如可以累加统计等。
- 使用 Set 做去重的计数统计。
- 使用 SortedSet 可以做排行榜等排序场景。
- 使用 getbit、setbit、bitcount 做大数据量的去重统计，在允许误差的情况下可使用 HyperLogLog。
- 使用 GEO 可以实现位置定位、附近的人。

以上场景基本上涵盖了 Redis 支持的各种存储结构。

- Key：可以是任意类型，但最终都会存储为 byte[]。
- String：简单的 (key,value) 存储结构，支持数据的自增、支持 BitSet 结构。
- Hash：哈希表数据结构，支持对 field 的自增等操作。

- list: 列表, 支持按照索引、索引范围获取元素以及 Pop、Push 等堆栈操作。
- Set: 集合, 去重的列表。
- SortedSet: 有序集合。
- HyperLogLog: 可对大数据进行去重, 有一定的误差率。
- GEO: 地理位置的存储结构, 支持 GEOHASH。

## 内存压缩

Redis 的存储是以内存为主的, 因此如何节省内存是非常关键的地方。

首先, key 越短越好, 可以采取编码或者简写的方式。如用户的笔记数目缓存 key 可以使用 `u:{uid}:n_count` 作为 key。同时, key 的数量也要控制, 可以考虑使用 hash 做二级存储来合并类似的 key 从而减少 key 的数量。

其次, value 也是越小越好, 尤其是在存储序列化后的字节时, 要选择最节省内存的序列化方式, 如 Kryo、Protobuf 等。

此外, Redis 支持的数据结构的底层实现会对内存使用有很大的影响, 如在缓存用户的头像时, 可以根据用户 ID 做分段存储, 每一段使用 hash 结构进行存储:

```
// 第1段 1~999
hset u:avatar:1 1 http://xxxx
hset u:avatar:1 2 http://xxxx

// 第2段 1000~1999
hset u:avatar:2 1000 http://xxxx
hset u:avatar:2 1999 http://xxxx
```

这样, 相比使用 String 存储, hash 底层会使用 ZipList 做存储, 极大地节省了内存使用。但这里需要注意的是, Redis 有一个 `hash-max-ziplist-entries` 参数, 默认值是 512, 如果 hash 中的 field 数目超过此值, 那么 hash 将不再使用 ZipList 存储, 而开始使用 HashTable。但是, 此值设置过大, 在查询的时候就会变慢。此外, 还有一个 `hash-max-ziplist-value` 参数, 默认是 64 字节, value 的最大字符串字节大小如果大于此值, 则不会使用 ZipList。

除了 hash 之外, 其他数据结构也有类似的内存编码变化, 使用的时候也需要注意, 如表 5-3 所示。

此外, 对于 list 来说, Redis 3.2 使用了新的数据结构 quicklist 来编码实现, 废弃了 `list-max-ziplist-value` 和 `list-max-ziplist-entries` 配置, 使用 `list-max-ziplist-size` (负数表示最大占用空间或者正数表示最大压缩长度) 和 `list-compress-depth` (最大压缩深度) 这两个参数进行配置。

表 5-3

数据结构	编码	条件
hash	ziplist	最大 value 大小 $\leq$ hash-max-ziplist-value && field 个数 $\leq$ hash-max-ziplist-entries
hash	hashtable	最大 value 大小 $>$ hash-max-ziplist-value
list	ziplist	最大 value 大小 $\leq$ list-max-ziplist-value && field 个数 $\leq$ list-max-ziplist-entries
list	linkedlist	最大 value 大小 $>$ list-max-ziplist-value
set	intset	元素都为整数 && 集合长度 $\leq$ set-max-intset-entries
set	hashtable	元素非整数类型
sortedSet	ziplist	最大 value 大小 $\leq$ zset-max-ziplist-value && 集合长度 $\leq$ zset-max-ziplist-entries
sortedSet	skiplist	最大 value 大小 $>$ zset-max-ziplist-value

## Redis Lua

一般情况下，Redis 提供的各种操作命令已经能够满足我们的需求。如果需要一次将多个操作请求发送到服务器端，可以通过 Jedis 客户端的 pipeline 接口批量执行。但如果有以下 3 种需求，就需要使用 Redis Lua。

- 需要保证这些命令作为一个整体的原子性。
- 这些命令之间有依赖关系。
- 业务逻辑除了 Redis 操作外还包括其他逻辑运算。

Redis 从 2.6 版本后内置对 Lua Script 的支持，通过 eval 或者 evalsha 执行 Lua 脚本。其脚本的执行具有原子性，因此适用于秒杀、签到等需要并发互斥且有一些业务逻辑的业务场景。如下：

```
String REDIS_SCRIPT_GRAB_GIFT =
    "local giftLeft = tonumber(redis.call('get',KEYS[1])) or 0;"
// 读取礼物剩余数量
    + "if(giftLeft <= 0) then return 0; end;" // 抢购失败
    + "redis.call('decr',KEYS[1]);" // 减少礼物数量
    + "return 1;";

...
Object grabResutl = jedis.eval(REDIS_SCRIPT_GRAB_GIFT, Lists.
newArrayList("test:gifts:" + giftId + ":left"),null);
...
```

使用 Redis Lua 需要注意以下几点。

- Lua 脚本里涉及的所有 key 尽量用变量，从外面传入，使 Redis 一开始就知道你要改变哪些 key，尤其是在使用 Redis 集群的时候。

- 建议先用 `SCRIPT LOAD` 载入 `script`，返回哈希值。然后用 `EVALHASH` 执行脚本，这样可以节省脚本传输的成本。
- 如果想从 Lua 返回一个浮点数，应该将它作为一个字符串（比如 `ZSCORE` 命令）。因为 Lua 中整数和浮点数之间没有什么区别，在返回浮点数据类型时会转换为整数。

## 数据失效和淘汰

如果某些数据并不需要永远存在，可以通过 `Expire` 设置其失效时间，让其在这段时间后被删除。这里设置了失效时间之后，可以通过 `SET` 和 `GETSET` 命令覆写失效期，或者使用 `PERSIST` 去掉失效期。需要注意的是，如果一个命令只是更新一个带生存时间的 `key` 的值，而不是用一个新的 `key` 值来代替它的话，那么生存时间不会被改变。如 `INCR`、`DECR`、`LPUSH`、`HSET` 等命令就不改变 `key` 的失效时间。此外，设置了失效期的 `key` 其 TTL 是大于 0 的，直至被删除会变为 -2，未设置失效期的 `key` 其 TTL 为 -1。

和大部分缓存一样，过期数据并非是立即被删除的。在 Redis 中，其采取的方式如下。

- 消极方法：主动 `get` 或 `set` 时触发失效删除
- 积极方法：后台线程周期性（每 100 毫秒一次）随机选取 100 个设置了有效期的 `key` 进行失效删除，如果有 1/4 的 `key` 失效，那么立即再选取 100 个设置了有效期的 `key` 进行失效删除。

这里需要注意的是，当使用主从模式时，删除操作只在 Master 端做，在 Slave 端做是无效的。

此外，当对 Redis 设置了最大内存 `maxmemory`，那么当内存使用达到 `maxmemory` 后，会触发缓存淘汰。Redis 支持以下几种淘汰策略。

- `volatile-lru`：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰，是 Redis 默认的淘汰策略。
- `volatile-ttl`：从已设置过期时间的数据集中挑选将要过期的数据淘汰。
- `volatile-random`：从已设置过期时间的数据集中任意选择数据淘汰。
- `allkeys-lru`：从数据集中挑选最近最少使用的数据淘汰。
- `allkeys-random`：从数据集中任意选择数据淘汰。
- `no-eviction`：禁止驱逐数据。

这里需要注意的是，Redis 中为了节省内存占用使用了整数对象池（即共享整数对象），但当淘汰策略为 LRU 时，由于无法对对象池的同一个对象设置多个访问时间戳，因此不会再使用整数对象池。

## 持久化

Redis 支持对内存中的数据进行持久化，包括两种实现方式。

### 1) RDB

RDB 是基于二进制快照的持久化方案，其在指定的时间间隔内（默认触发策略是 60 秒内改了 1 万次或 300 秒内改了 10 次或 900 秒内改了 1 次）生成数据集的时间点快照（Point-In-Time Snapshot），从而实现持久化。基于快照的特性，会使其丢失一些数据，比较适用于对 Redis 的数据进行备份。此外，RDB 进行时，Redis 会 fork() 出一个子进程，并由子进程来遍历内存中的所有数据来进行持久化。当数据集比较庞大时，由于 fork 出的子进程需要复制内存中的数据，因此这个过程会非常耗时，会造成服务器停止处理客户端，停止时间可能会长达 1 秒。

可配置 RDB 对数据进行压缩存储，支持字符串的 LZF 算法和 string 形式的数字变回 int 形式。

### 2) AOF

AOF 是基于日志的持久化方案，记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。这些命令全部以 Redis 协议的格式来保存（纯文本文件），新命令会被追加到文件的末尾。此外，为了避免 AOF 的文件体积超出保存数据集状态所需的实际大小，Redis 在 AOF 文件过大时会 fork 出一个进程对 AOF 文件进行重写。AOF 这种方案，默认是每隔 1 秒进行一次 fsync（将日志写入磁盘），因此与 RDB 相比，其最多丢失 1 秒的数据，当然如果配置成每次执行写入命令时 fsync（非常慢），甚至可以避免任何数据的丢失。但其文件的体积明显大于 RDB，将日志刷到磁盘和从 AOF 恢复数据的过程也是慢于 RDB 的。

如果想要保证数据的安全性，建议同时开启 AOF 和 RDB，此时由于 RDB 有可能丢失文件，Redis 重启时会优先使用 AOF 进行数据恢复。

此外，可以通过 save 或者 bgsave 命令来手动触发 RDB 持久化，通过 bgrewriteaof 触发 AOF 重写。如此可以将 RDB 或者 AOF 文件传到另一个 Redis 节点进行数据迁移。

需要注意的是，如果通过 kill -9 或者 Ctrl+C 来关闭 Redis，那么 RDB 和 AOF 都不会被触发，这样会造成数据丢失，建议使用 redis-cli shutdown 或者 kill 优雅关闭 Redis。

## 分布式

Redis 本身对分布式的支持有如下这 3 种。

### 1) Master-Slave

简单的主从模式，通过执行 `slaveof` 命令来启动，一旦执行，Slave 会清掉自己的所有数据，同时 Master 会 `bgsave` 出一个 RDB 文件并以 Client 的方式连接 Slave 发送写命令给 Slave 传输数据。

Redis 还提供了 Redis Sentinel 做这种方案的 fail-over，能够对 Redis 主从复制进行监控，并实现主挂掉之后的自动故障转移。

首先，Sentinel 会在 Master 上建一个 pub/sub Channel，通告各种信息。所有 Sentinel 通过接收 pub/sub Channel 上的 +Sentinel 的信息发现彼此（Sentinel 每 5 秒会发送一次 `__sentinel__:hello` 消息）。然后，Sentinel 每秒会对所有 Master、Slave 和其他 Sentinel 执行 ping，这些 Redis-Server 会响应 +PONG、-LOADING 或者 -MASTERDOWN 告知其存活状态等。如果一台 Sentinel 在 30 秒内没有收到 Master 的应答，会认为 Master 已经处于 SDOWN 状态，同时会询问其他 Sentinel 此 Master 是否 SDOWN，如果 quorum 台 Sentinel 认为 Master 已经 SDOWN，那么就认为 Master 是真的挂掉（ODOWN），此时会选出一个状态正常且与 Master 的连接没有断开太久的 Slave 作为新的 Master。

Redis Sentinel 提供了 notify 脚本机制可以接收任何 pub/sub 消息，以便于发出故障告警等信息；提供了 reconfig 脚本机制在 Slave 开始提升成 Master、所有 Slave 都已指向新 Master、提升被终止等情况下触发对此类脚本的调用，可以实现一些自定义的配置逻辑。

### 2) Redis Cluster

Redis 3.0 后内置的集群方案。此方案是没有中心节点的，每一个 Redis 实例都负责一部分 slot（存储一部分 key），业务应用需要通过 Redis Cluster 客户端程序对数据进行操作。客户端可以向任一实例发出请求，如果所需数据不在该实例中，则该实例引导客户端去对应实例读/写数据。Redis Cluster 的成员管理（节点名称、IP、端口、状态、角色）等，都通过节点之间两两通信、定期交换并更新。这是一种比较重的集群方案。

Redis 的集群方案除了内置的 Redis Cluster 外，很多公司都采用基于代理中间件的思路做了一些实现，Twemproxy、Codis 是其中用得比较多的软件。相比官方的集群方案，其使用方式和单点 Redis 是一模一样的，原有的业务改动很少（个别命令会不被支持），且其数据存储和分布式逻辑是分离的，便于扩展和升级。

### 3) 在客户端做分片

除了上述集群方案外,在客户端做分片也是一种常用的 Redis 集群实现方式,不依赖于第三方分布式中间件,实现方法和代码都自己掌控,相比代理方式少了中间环节。但是此方式数据迁移、合并等都不够灵活,建议慎用。

## 使用提示

### 1) Redis 数据操作。

- 不要在大数据量线上环境中使用 `keys` 命令,这很容易造成 Redis 阻塞。
- 尽量使用 `mset`、`hset` 等做批量操作,以节省网络 I/O 消耗。
- 在 Java 中使用 Jedis 的 `pipeline` 一次执行多条相互没有依赖关系的命令可以节省网络 I/O 的成本,但 `pipeline` 和事务不同,其只是一种批量写、批量读的多命令流水线机制,Redis 服务器并不保证这些命令的原子性。
- 使用 Redis 的事务命令 (`multi`、`exec`、`discard`),其事务级别类似于 `Read Committed`,即事务无法看到其他事务未提交的改动。还可以使用 `watch` 对某一个 `key` 做监控,当 `key` 对应的值被改变时,事务会被打断,能够达到 CAS 的效果。
- 使用 `list` 做队列时,如果需要 `ACK`,可以考虑再使用一个 `SortedSet`,每次队列中 `Pop` 出一个元素则按照访问时间将其存储到 `SortedSet` 中,消费完后进行删除。
- 对大集合键数据的删除(元素非常多的 `hash`、`list`、`set`、`sortedSet`)避免使用 `del`,会造成 Redis 阻塞。
  - `hash`: 通过 `hscan` 命令,每次获取一部分字段,再用 `hdel` 命令,每次删除一个字段。
  - `list`: 使用 `ltrim` 命令每次删除少量元素。
  - `set`: 使用 `sscan` 命令,每次扫描集合中一部分元素,再用 `srem` 命令每次删除一个键。
  - `sortedSet`: 使用 `zremrangebyrank` 命令,每次删除顶部 100 个元素。
- 在 Java 开发中一般选择直接使用 Jedis 即可。如果需要诸如分布式锁、主从等分布式特性或者应用层级的 Redis 操作封装,可以选择使用 `Redisson` 库来操作 Redis。

### 2) 配置与监控。

- 可以通过 `monitor` 命令监测 Redis 上命令执行的情况。
- 由于 Redis 自身单线程的原因,切忌慢查询,会阻塞住整个 Redis,可以通过 `slowlog get` 来查看慢查询日志。

- 设置 Redis 最大内存，以防内存用爆。
- 使用 redis-rdb-tools 对 rdb 文件进行分析，如每条 key 对应的 value 所占的大小，从而做量化分析。
- 可以使用 Redis Sampler，以统计 Redis 中的数据分布情况。

### 5.3.3 缓存设计的典型方案

在使用缓存系统的时候，还需要考虑缓存设计的问题，重点在于缓存失效时的处理和如何更新缓存。

缓存失效是在使用缓存时不得不面对的问题。在业务开发中，缓存失效时由于找不到整个数据，一般会出于容错考虑，从存储层再进行查询，如果有则放入缓存。如果查找的数据在存储层根本就不存在，缓存失去意义，还会给后端服务带来巨大的请求压力，会进一步引起雪崩效应。这种现象又被称为缓存穿透。

目前常用的解决缓存穿透问题的方案如下。

1) 在底层存储系统之上加一层布隆过滤器，将所有可能存在的数据哈希到一个足够大的 BitMap 中，一个一定不存在的数据会被这个 BitMap 拦截掉，从而避免了对底层存储系统的查询压力。

2) 如果数据在存储层查询也为空，那么对此空结果也进行缓存，但要设置合适的失效时间。

更进一步地，解决缓存穿透问题其实和缓存的更新机制是相关的。缓存更新的常用 3 种模式如下。

- **Cache Aside Pattern**: 应用程序以数据库为准，失效则从底层存储更新，更新数据先写入数据库再更新缓存。这是最常用的缓存更新模式。
- **Read/Write Through Pattern**: 以缓存为准，应用只读 / 写缓存，但是需要保证数据同步更新到数据库中。
- **Write Behind Caching Pattern**: 以缓存为准，应用只读 / 写缓存，数据异步更新到数据库，不保证数据正确写回，会丢数据。可以采用 Write Ahead Logging 等机制避免丢数据。

如上，在缓存失效时采用何种策略去更新缓存，直接决定了能否解决缓存穿透的问题。Cache Aside Pattern 中缓存失效从底层存储更新，无法避免缓存穿透的问题。基于以上 3 种模式，采用下面更为细化的更新机制可以在一定程度上避免缓存穿透的问题。



- 缓存失效时，用加锁或者队列的方式单线程 / 进程去更新缓存并等待结果。
- 缓存失效时，先使用旧值，同时异步（控制为同时只有一个线程 / 进程）更新缓存，缓存更新失败则抛出异常。
- 缓存失效时，先使用旧值，同时异步（控制为同时只有一个线程 / 进程）更新缓存，缓存更新失败则延续旧值的有效期。
- 数据写入或者修改时，更新数据存储后再更新缓存。缓存失效时即认为数据不存在。
- 数据写入或者修改时，只更新缓存，使用单独线程周期批量刷新缓存到底层存储。缓存失效时即认为数据不存在。这种方案不能保障数据的安全性，有可能会丢数据。
- 采用单独线程 / 进程周期将数据从底层存储放到缓存中。缓存失效时即认为数据不存在。这种方案无法保证缓存数据和底层存储的数据强一致性。

如果一开始设计缓存结构的时候注意切分粒度，把缓存力度划分得细一点，那么缓存命中率相对会高，也能在一定程度上避免缓存穿透的问题。

此外，还可以在后端做流量控制、服务降级或者动态扩展，以应对缓存穿透带来的访问压力。

## 5.4 搜索引擎——Elasticsearch

前几节讲过的 MySQL 和 MongoDB 都有全文检索的功能，但如果真的有全文检索的需求，搜索引擎才是最好的方案。目前 Java 开发中最常用的搜索引擎是 Solr 和 Elasticsearch。

其中，Solr 是 Apache 旗下的开源搜索引擎，基于 Lucene 开发。支持层面搜索、命中醒目显示和多种输出格式，包括 XML/XSLT 和 JSON 格式，是之前用得比较广泛的搜索引擎。但近几年 Elasticsearch 已经开始逐步取代 Solr，正逐渐成为主流的搜索引擎选型。因此，本节主要针对 Elasticsearch 介绍。

Elasticsearch 是一个分布式、RESTful 的搜索以及分析服务器，其和 Solr 一样也是基于 Lucene 的。其主要优点如下。

- 轻量级，下载后一条命令即可启动。
- 可以存储任意结构的 JSON 对象，是无模式的。
- 可以使用不同的 index 参数创建不同的索引文件，实现多索引文件支持。
- 天然具有分布式特性，可以自动发现 Elasticsearch 节点。

- 不仅仅是搜索引擎，还有数据分析、聚合、可视化等特性。
- 可以把 Elasticsearch 直接当作 NoSQL 数据库使用，存储 JSON 文档。
- 为各种操作都提供了 RESTful 接口。
- 有强大的聚合查询功能。

Elasticsearch 目前已经到了 5.x 版本，但由于新版本改动较大，目前用得还不太多。本节使用的 Elasticsearch 为 2.3.3 版本。

### 5.4.1 开源全文检索库——Apache Lucene

Apache Lucene 是一个 Java 开源全文检索库，是很多搜索引擎的检索实现机制。其中有几个关键概念介绍如下。

- 文档：Document，索引与搜索的主要数据载体。
- 字段：Field，文档的一个片段。
- 词项：Term，搜索时的一个单位，一般为一个词。
- 词条：Token，词项在字段中的一次出现。

Lucene 将索引信息组织为词典（Term Dictionary）。每一个条目由词项和该词出现过的文档集合组成，这样就能够通过词反向查询文档信息，如表 5-4 所示。

表 5-4

词项	文档
篮球	[1,2]
足球	[1,4,5]
乒乓球	[7]

为了加快在词典中查找词项的速度，Lucene 对词典做了前缀索引，版本 4.0 后使用 FST（finite state transducer）这个数据结构将其保存在内存中，会在每次打开索引时全量装载到内存中。

图 5-5 即一个 FST（通过 <http://examples.mikemccandless.com/fst.py> 网址生成），是一个有向无环图，依次插入的单词为 basketball、football、pingpong，之后可以很方便地查询这些单词。其用途和 HashMap、Tries 树基本相同，但是 FST 非常省内存。

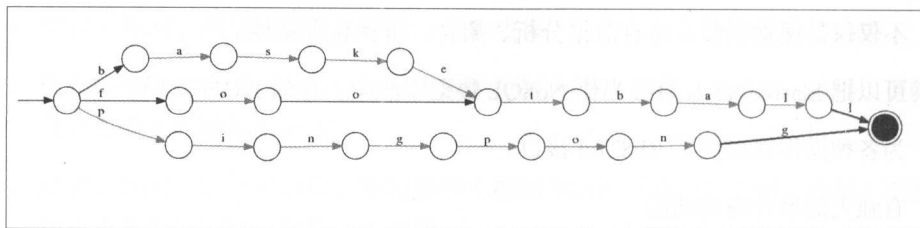


图 5-5

Lucene 中的索引由多个 segment 段组成，每一段只被创建一次但被查询多次。索引期间，段不可以被修改。因此，文档的删除也仅仅是把信息保存在一个单独的文件中，段本身并没有改变。Lucene 会对多个段进行合并，称为 segments merge。目的是为了减少 segment 的数目，把小的 segment 合并为大的 segment，减少搜索的 segment 数。此外，merge 也会把被删除的信息清理掉。合并可以强制执行，也可以由 Lucene 的内在机制自动执行，但这个合并过程非常消耗 I/O，不建议强制执行。

倒排索引中的 key 是文档中的词，这些词都是通过分析器来对文档进行分析得到的。分析器包括：分词器、过滤器和字符映射器。

- **分词器**：将文本切割成词条，输出词条流。
- **过滤器**：用来处理词条流中的词条，移除、修改词条。
- **字符映射器**：做文本预处理工作，比如去除文本的 HTML 标签。

Lucene 提供了一套查询语言用于对文档的字段进行查询，支持通配符查询和模糊查询。

基于 Lucene 提供的这些强大的全文索引功能，Solr、Elasticsearch 在其上封装出了独立的服务，增加了分布式、容错、分词器实现、数据持久化等特性和组件，这就形成了搜索引擎。

## 5.4.2 关键概念

### 1) 节点、集群。

每一个运行的 Elasticsearch 服务实例叫作节点（Node）。多个 Elasticsearch 实例以集群（Cluster）方式运行，以一个整体对外提供服务，能够容错并提供巨大的数据量支持。Elasticsearch 天然支持集群，配置非常简单。

### 2) 索引。

索引（index）是 Elasticsearch 存储数据的地方，类比于关系型数据库的 database，是被索引文档的集合，可以向索引写入 / 读取文档。此外，Elasticsearch 提供了索引的别名（Alias）机制，一个索引可以有多个别名，一个别名可以对应多个索引。

### 3) 文档。

文档 (document) 是 Elasticsearch 中的主要实体, 没有固定的模式, 可以看作 JSON 对象。所有的搜索最终都是对文档的搜索, 文档由字段构成, 每个字段有一个名字和一个或者多个字段值。

### 4) 分片。

Shard, 是 Elasticsearch 提供分布式搜索的基础。Elasticsearch 将一个完整的 index 分成若干部分存储在相同或者不同的节点上, 组成 index 的部分即为 Shard, 每一个 Shard 都是一个 Lucene 实例。每个 Shard 都有一个编号, Doc 落入哪个 Shard 默认与文档的 ID 有关, 也可以通过配置路由 (route) 来自定义。

### 5) 副本。

Elasticsearch 会为索引的每一个 Shard 创建冗余副本 (Replica), 用来提高系统的容错性, 并可以自动对搜索请求进行负载均衡, 提高 Elasticsearch 的查询效率。Elasticsearch 支持在任意时间点添加或移除副本, 可随时调整副本数量。

### 6) Recovery。

Elasticsearch 在有节点加入或退出时会根据机器的负载对索引分片进行重新分配, 挂掉的节点重新启动时也会进行数据恢复。

### 7) 网关。

Elasticsearch 默认先把索引存放到内存中, 当内存满了时再持久化到本地硬盘。网关对索引快照进行存储, 当这个 Elasticsearch 集群关闭再重新启动时, 就会从网关中读取索引备份数据, 此外还存储了集群状态、索引设置的各种信息。

### 8) Zen Discovery。

Elasticsearch 的自动发现节点机制, 也可以叫作共识系统。Elasticsearch 是一个基于 P2P 的系统, 它先通过广播寻找存在的节点, 再通过多播协议来进行节点之间的通信, 同时也支持点对点的交互, 并进行 Master 的选举。

## 5.4.3 查询的优化

Elasticsearch 中的节点都是对等的, 因此索引请求可以随便发到哪个节点上, 但之后此节点经过信息查询后, 都把请求分发到包含对应编号的主 Shard 的节点上执行操作, 操作成功后再在备 Shard 上执行操作。

Elasticsearch 的更新和查询的底层过程如下。

- 当索引增加、更新时，请求的结果都先存入 indexing buffer。当 buffer 满了或者到了时间周期，就会将 buffer 中的内容写入磁盘上的 segment（对 Lucene index reader 调用了 reopen，并没有 fsync 保证持久化）。
- 当有删除请求到来时，被删除的文档 ID 会被记录到一个单独的文件中。
- 每次查询的时候，会扫描内存、硬盘中的 Doc，然后过滤掉 deleted 文件中的 Doc。

依照上述过程 indexing buffer 生成 segment 默认的周期是 1 秒，使得被索引的文档可以被近实时地搜索到，达到准实时读取。但如果有意外，有可能会造成数据丢失。Elasticsearch 提供了 translog 来保证一定的数据完整性，每一个 Shard 都会有自己的 translog。任一索引和删除操作在被 Lucene 处理后都会写入 translog。而为了防止 translog 过大，会有一个 flush 过程，此过程会触发 Lucene 的 commit 将内存中的文档存入磁盘并开始新的 translog，这非常耗费时间和资源，因此控制 flush 发生的时机是影响索引的关键点之一。相关配置参数如下。

- index.translog.flush\_threshold\_size: translog 一旦达到此值就会进行一次 flush，可以调大此值甚至关闭，手动进行 translog flush。
- index.translog.flush\_threshold\_ops: translog 数据达到此值就会进行一次 flush，可以调大此值甚至关闭，手动进行 translog flush。
- index.translog.flush\_threshold\_period: 距离上一次 flush 的时间间隔阈值。

在打开的索引中，只要满足以上任意一条 translog 不为空即可触发 flush。此外，还有两个参数也关系着 Elasticsearch 的性能。

- index.translog.interval: translog 的提交周期，默认为 5 秒。
- index.refresh\_interval: 指多久进行一次数据刷新使得索引可以被使用。可以设置为 -1 关闭该功能，手动进行 refresh。

还有以下因素也会影响索引质量，从而影响检索速度。

#### 1) 分片数目。

分片数目过多会导致检索时打开的文件较多、多台服务器之间通信，过少则单个分片索引过大，检索速度慢。分片数根据数据总量 / 单片的数据数目来计算，单片的数据数目需要通过单节点、单索引来进行测试，一般不超过 10GB 即可：

```
PUT /my_index
{
```

```

"settings": {
  "number_of_shards" : 1
}
}

```

## 2) 副本数。

如果有副本存在，那么索引过程会同步到副本中。在对索引的安全性没有那么高的要求的情况下，可以在索引过程中将副本数设置为 0，待索引完成后再改回去，这样可以在一定程度上提高索引效率：

```

PUT /my_index
{
  "settings": {
    "number_of_replicas" : 0
  }
}

```

## 3) 分词。

分词选择的词库要适量，并非越多越好，词库越多，词表越大，那么分的词就会变多，从而索引也会变大。根据业务场景选择与特征相关的词库，能够使词表变小，索引大小也会减小许多。

## 4) 索引段。

每次搜索请求会搜索所有的段，因此 segment 的数目过多会导致搜索时读取的文件数过多，影响搜索效率。虽然有对 segment 自动做合并的机制，但在可能的情况下把 segment 的数目设置为 1 可以更好地提高检索速度：

```

curl -X POST 'http://localhost:9200/my_index/_forcemerge?max_num_segments=1'

```

上述命令即可将索引的段合并为一个。

# 5.4.4 内存的使用优化

Elasticsearch 是一个 Java 应用，因此内存和垃圾回收是影响 Elasticsearch 性能的关键因素。

Elasticsearch 的倒排索引是先在内存在中生成的，然后定期以 segment file 的形式刷到磁盘中。这样每个 segment 都会有一些索引数据存储在 heap 里，如词项索引。segment 越多，占用的 heap 也越多，且无法进行垃圾回收，因此 Elasticsearch 的数据存储并非仅仅消耗磁盘空间，当 segment memory 占用过大时就需要考虑删除、归档数据或者扩容。

官方建议设置的 heap size 不要超过系统可用内存的一半，heap 以外的内存操作系统会用来 cache 数据。JVM 的 xms 和 xmx 也要设置为和 heap 一样大，避免动态扩展。

为了减小 segment memory 的占用空间，有以下方法。

- 删除不用的索引。
- 定期对不再更新的索引做 force merge，即对 segment file 强制做合并，这样可以节省大量内存占用。

此外，为了防止内存频繁 swap 影响 Elasticsearch 性能，有以下两种方式。

- 关闭系统的 swap 功能 `sudo swapoff -a`，但由于会关闭 Linux 系统的 swap 功能，所以最好结合机器的情况来选定是否执行此配置。
- 配置 `bootstrap.mlockall: true`，让 JVM 锁住内存，同时需要运行 Elasticsearch 的 Linux 用户有锁定内存区域的权限。

Elasticsearch 的启动用户为 `esUser`，修改 `/etc/security/limits.conf` 文件：

```
esUser soft memlock unlimited
esUser hard memlock unlimited
```

### 5.4.5 开源日志管理方案——ELK

ELK，即 Elasticsearch + Logstash + Kibana，是一套开源的日志管理方案，可以构建日志集中分析平台。ELK 的一般架构如图 5-6 所示。

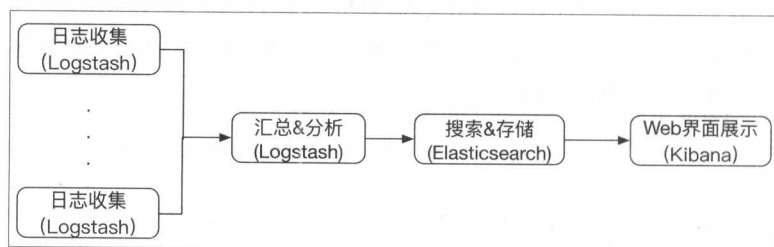


图 5-6

- **Logstash**：多个独立的 Agent 负责收集不同来源的数据，一个中心 Agent 负责汇总和分析数据。
- **Elasticsearch**：用于存储最终的数据，并提供搜索功能。
- **Kibana**：提供一个简单、丰富的 Web 界面，数据来自 Elasticsearch，支持各种查询、统计和展示。

还可以在远程 Logstash 和中心 Logstash 之间加入 Redis、Kafka 等中间代理层作为缓冲和中间存储，提高系统性能和可靠性。

## 使用提示

- Elasticsearch 性能体现在分布式上，一个生产系统建议至少 3 个节点以上。
- 倒排词典的索引是常驻内存的，需要监控数据节点上 segment memory 的使用情况。
- 要定期删除不再使用的索引并做好冷数据的迁移。
- 如果不使用 `_all` 字段，那么关闭这个属性，否则在创建索引和增大索引大小的时候会使用更多的 CPU 资源。
- 避免返回大量结果集的搜索和聚合，可以采用 scan 和 scroll API 来补充缺失数据。
- 善用 bool query，类似于之前版本的 filtered query，可以在进行复杂的倒排算法之前先减小计算空间。
- 使用批量查询和批量读取减少网络 I/O。
- 可以使用多个别名命名同一个 index，然后针对不同的别名做不同的操作权限控制。
- Elasticsearch 的 Java 客户端有两种类型：NodeClient 是集群中的一个节点，会同步路由等信息，影响启动速度。TransportClient，仅仅作为一个客户端，启动速度比较快，但是其不知道集群的任何信息，因此查询时需要先发送请求到某个节点，然后由 Elasticsearch 转发到文档所在的节点做处理，查询需要消耗更多资源。



# 第 6 章

## 数据通信

数据存储在各个系统中，系统虽然大部分功能都是自身实现的，但是很多时候也需要依赖第三方服务，以及提供服务给第三方和客户端、前端。这时候就需要数据通信，即如何将数据从一个系统转移到另一个系统。

需要注意的是，这里讲的数据通信指的主要是系统之间的数据通信。因此，从底层介质来讲，基本都是基于网络进行的。而网络传输则主要通过 TCP 和 HTTP 两种协议。

其中，TCP 是比较底层的传输协议，基于此协议需要自己做很多开发工作。而 HTTP 是 TCP 之上的应用协议，基于 HTTP 的数据传输机制实现较容易，但无法针对特殊场景做底层的优化，性能上也不如 TCP 协议的数据传输机制。

基于以上协议，目前常用的系统间数据传输方案主要包括以下几种。

- RESTful：符合 REST（Representational State Transfer）架构风格的设计，主要指 API 的设计。
- RPC：远程过程调用，可以基于 TCP 协议，也可以基于 HTTP 协议。
- 消息中间件：利用消息队列，作为数据传输的介质，基本上都是基于 TCP 协议的。

### 6.1 RESTful 架构风格

REST，全称表现层状态转移（Representational State Transfer），指的是资源在网络中以某种表现形式进行状态转移，是一种架构风格。其描述的是网络中 Client 和 Server 的一种交互形式。简单来说就是用 HTTP URL 来定位资源，用 HTTP 的各种 method 来描述操作。其关键的 3 个概念如下。

- Resource: 资源, 主要指的是数据。
- Representational: 数据的表现形式, 如 JSON、XML、HTML 等。
- State Transfer: 状态变化, 通过 HTTP method 来描述。

REST 经常被用来规范 API 的设计以及数据传输的格式, 可以统一给各种客户端提供接口, 包括 Web、iOS、Android 和其他的服务。REST 不需要显式的前端页面, 只需要按照格式返回数据即可。符合 REST 风格的 API 被称为 RESTful API, 符合 RESTful 规范的架构被称为 RESTful 架构, 如图 6-1 所示。

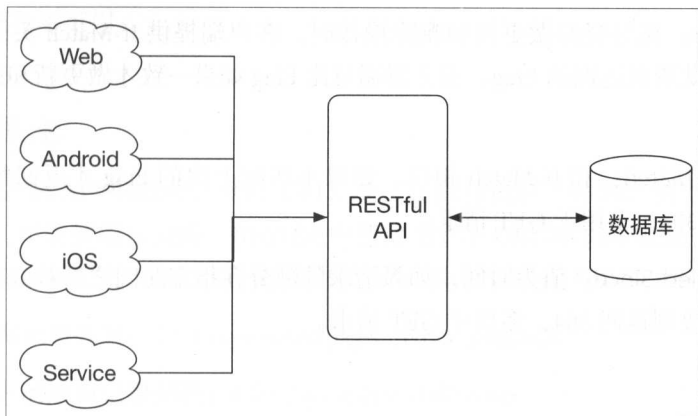


图 6-1

### 6.1.1 支持的操作

RESTful 基于 HTTP 协议, 其主要依赖于 HTTP 协议的几种 method 来表示 CRUD( create、read、update 和 delete, 即数据的增删查改) 操作。

- GET: 从服务器上获取资源。
- POST: 创建新的资源。
- PUT: 更新服务器资源。
- DELETE: 删除服务器资源。

这里需要注意以下几点。

- GET、PUT 和 DELETE 应该是幂等的, 即在相同的数据和参数下, 执行一次或多次产生的效果是一样的。
- 对于 POST 和 PUT 操作, 则应该返回最新的资源, 删除操作则一般不必要。

- 所有的操作都是无状态的，即所有的资源，都可以通过 URL 定位，这个定位与其他资源无关，也不会因为其他资源的变化而改变。

除了上述方法外，还有一个 PATCH 方法也用于更新资源的部分属性，但用得并不多，一般用 POST 即可。

此外，HTTP 1.1 的几个头部也是应该注意的。

- **Accept:** 客户端要求服务器返回什么样表现形式的数据。RESTful API 需要根据此头部返回合适的数据。
- **If-Match:** 在对资源做更新和删除操作时，客户端提供 If-Match 头，值为服务器端上次对此资源返回的 Etag，服务器端对比 Etag 如果一致才做更新和删除，否则返回 412。
- **If-None-Match:** 和 If-Match 相反，如果不匹配上次的 Etag 才返回数据，匹配的话则返回 304，多用于 GET 请求。
- **If-Modified-Since:** 值为时间，如果请求的部分在指定时间之后被修改则请求成功，未被修改则返回 304，多用于 GET 请求。

## 6.1.2 返回码

HTTP 本身已经提供了很多 StatusCode 来表示各种状态。RESTful 接口需要遵循这些定义，返回合适的状态码和数据。当然，如果是内部使用，统一返回 200，在返回数据里自定义一套状态码也是可以的。

HTTP 的状态码大体分为以下几个区间。

- 2XX: 请求正常处理并返回。
- 3XX: 重定向，请求的资源位置发生变化。
- 4XX: 客户端发送的请求有错误。
- 5XX: 服务器端错误。

在自己设计返回码的时候最好也遵循此范围设计，以下是其中几个常用的状态码。

- 200: 表示请求成功。
- 301: 资源已经永久迁移到新的地址，新的 URL 会在响应头中返回。
- 302: 资源临时被迁移到新的地址，新的 URL 会在响应头中返回。

- 304: 表明资源未改变。主要配合请求头中的 If-None-Match 和 If-Modified-Since 使用。
- 400: 错误请求, 表示请求中有语法错误。
- 401: 请求的资源需要认证, 请求没有提供认证信息或者认证错误。
- 403: 资源被禁止访问。
- 404: 资源不存在。
- 502: 错误的网关, 通常指作为代理的服务器无法收到远程服务器的正确响应。
- 503: 服务不可用。

### 6.1.3 资源概念

资源是 RESTful API 的核心, 其以 URI (统一资源标识符) 标识, 而 URL 则不仅能够标识一个资源, 还能够定位资源。RESTful 中使用 HTTP URL 标识并定位一个资源。原则上只使用名词来指定资源, 而且推荐使用复数。下面以对记事的 CRUD API 的设计为例。

- 获取所有记事列表: `GET /api/notes?page=1&per_page=20`。
- 获取某人的所有记事列表: `GET /api/users/{uid}/notes`。
- 获取标记为星的记事: `GET /api/users/{uid}/notes?star=1`。
- 创建记事: `POST /api/notes`。
- 删除某一个记事: `DELETE /api/notes/{note_id}`。
- 更新某一个记事: `PUT /api/notes/{note_id}`。

可知,

- 资源分为单个资源和资源集合, 尽量使用复数来表示资源, 单个资源通过添加 ID 等标识符来表示。
- 资源使用嵌套结构, 类似于目录路径的方式, 可以体现出彼此之间的关系。
- 一个资源可以有不同的 URL, 如上可以获取所有的记事列表, 也可以获取某人的所有记事列表。
- 对于 GET 方法, 一定不能设计为可以改变资源的操作, 如 `get /api/deleteNote?id=xx`。
- URL 是对大小写敏感的, 尽量使用小写字母, 单词间用下画线连接。

- 使用 Query 参数来控制返回结果，如上面返回星标记事的接口。此外，像排序方向、排序使用的字段等都是可以放在 Query 参数中的。
- 分页参数使用 Query 参数 (page、per\_page) 控制，在返回数据中返回当前页、下一页、上一页、总页数等分页相关信息。

如果需要区分版本号，可以放在路径中，如 /api/v2/\*\*，也可以放在 header 的 Accept 字段或者 Query 参数中：

```
Accept: version=2.0;...
```

对于一些很难设计为 CRUD 操作的 URL，如登录、送礼物等，有以下处理方式。

- 使用 POST，如 POST /api/login。
- 把动作转换成资源：登录就是创建了一个 Session 或者 Token，那么就可以设计为 POST /api/sessions。

此外，对于数据的提交格式和返回格式，目前以 JSON 格式为主，其可读性、紧凑性、多语言支持都较好；数据提交的方式也应该使用 application/JSON 的内容格式并在 body 里放置 JSON 数据：

```
...
Content-type: application/json
Accept: application/json
...

{
    'title': 'xxx',
    'content': 'xxx'
    ...
}
```

### 6.1.4 数据的安全保障

HTTP 本身不对数据做任何安全处理，因此建议首先从根本上使用 HTTPS 加强数据的安全性。此外，这里的安全性还要保证数据的完整性；保证接口的授权访问，保证接口只提供给授权过的应用访问以及过滤掉不必要的请求；保证数据的授权访问，只允许资源拥有者删除、更新自己的资源。

#### 数据的完整性

数据完整性主要是指在对数据进行修改时，保证要修改的数据与服务器数据是一致的。可以通过 Etag 这个 HTTP 中的头部字段来解决。

Etag 表示的是资源的唯一版本号，在请求资源时，RESTful API 应该返回资源数据以及资源的 Etag。API 请求方修改资源时应该提交 If-Match 头，这样服务器通过对比 Etag 可以防止数据被错误修改，类似于并发中的 CAS 原理。但是要绝对保证数据的完整性，还需要配合严格的并发控制才能做到。

## 接口访问控制

接口访问控制可以保证接口的授权访问，拒绝不合法的请求。可以通过以下几种方式进行结构访问控制。

- 在 Request header 中添加特殊的标识符，如果有不含此 header 的请求则直接拒绝。这样可以做简单的接口访问控制。
- 过滤 Request query 和 body，做白名单验证，即只允许出现哪些参数，如果有非法参数，可以抛弃或者直接拒绝请求。

上面只是比较简单的接口访问控制策略，无法彻底拒绝未授权的请求。我们可以通过为每一个授权应用分配 app\_secret（私有的、不公开），在访问时对请求进行签名验证的方式实现，以更为严格的接口访问控制，这种方法也被叫作 HMAC。一个请求签名生成的例子如下：

```
app_sign = MD5(METHOD & PATH & timestamp & app_secret)
```

其中，METHOD 指的是此次请求的方法，PATH 指 URL 中的 path 部分，timestamp 是请求时间戳，app\_secret 是分配请求方的私钥，此外还有一个分配给请求方的 app\_id。这样，app\_id、timestamp、app\_sign 随着请求一起发送（可以作为 Query 参数，也可以作为 header），服务器接收到请求后使用同样的算法计算出 app\_sign 进行对比，如果相同则正常请求，否则返回 401 Unauthorized。由此既可以保证接口的授权访问，还能够基于时间戳防止重放攻击。当然，app\_sign 的生成算法可以加入更多的因子，如 request\_body、query 等。但需要注意的是，这个算法越复杂，对接口的性能影响就越大，需要做权衡。

## 数据的授权访问——OAuth

数据的授权访问其实也是接口访问控制的一部分，主要关注点在于对资源的操作权限做控制。基于 HTTP 做授权访问的核心就是验证一个请求是否是合法用户发起的，主要的有 HTTP Basic Auth、OAuth 等。其中 Basic Auth 会把用户的用户名和密码直接暴露在网络中，并不安全，因此 RESTful API 主要使用 OAuth 做数据的授权访问控制。

OAuth 2.0 的验证流程如图 6-2 所示。

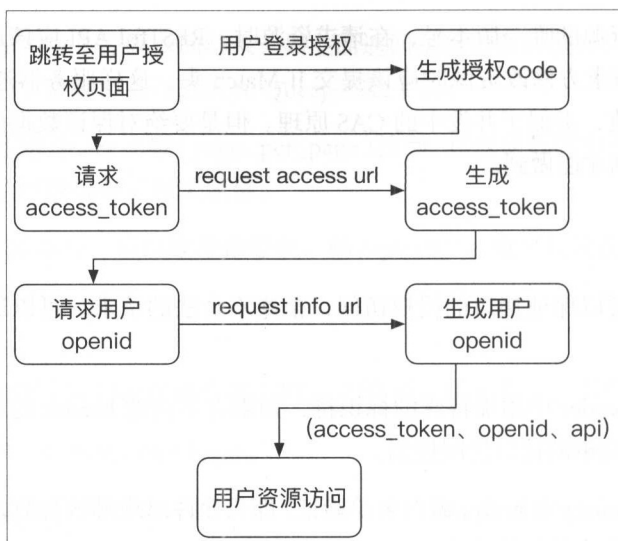


图 6-2

- 得到授权码 code。
- 使用授权码换取 access\_token 和 refresh\_token，通常 refresh\_token 比 access\_token 有效期长。
- 使用 access\_token 获取用户 openid。
- 使用 access\_token 和用户 openid 调用用户授权接口。
- 使用 refresh\_token 获取新的 access\_token。

当然，如果是提供给内部应用的 API，可以做适当简化，比如用户登录直接返回 access\_token，凭借此 access\_token 调用授权接口即可。

### 6.1.5 请求的限流

RESTful API 应该有限流机制，否则会造成 API 被滥用甚至被 DDOS 攻击。可以根据不同的授权访问做不同的限流，以减少服务器压力。

限流的情况可以通过下面几个头部字段返回给请求方。

- X-RateLimit-Limit：用户每个小时允许发送请求的最大值。
- X-RateLimit-Remaining：当前时间窗口剩下的可用请求数目。

- X-RateLimit-Reset: 在时间窗口重置的时候, 到这个时间点可用的请求数量就会变成 X-RateLimit-Limit 的值。

对于未登录的用户根据 IP 或者设备 ID 来限流, 对于登录用户根据用户标识。对于超过流量的请求, 返回 403 forbidden 或者 429 Too many requests 都可以。

### 6.1.6 超文本 API

RESTful 还有一个非常关键的特性就是超文本 API (Hypermedia API), 指的是服务器需要在每一个 API 接口的返回结果中都提供与下一步操作相关的资源链接, 客户端借助这些实现表现层转移状态。这种设计也被称为 HATEOAS (Hypermedia as the Engine of Application State)。

除此之外, 这样做还能够让客户端和服务端解耦, 客户端只需要依次遍历返回结果中的超链接就能完成一系列业务逻辑; 当服务端做了业务逻辑改动后, 也只需要修改服务器返回的资源链接。

### 6.1.7 编写文档

RESTful API 一般是对接第三方的, 因此文档说明非常必要。对每一个接口都详细地说明参数含义、数据返回格式和字段意义并举出实际的例子都是非常关键的。

在 Java Web 开发中, 我们可以使用 Swagger UI + Spring Fox 来基于注释生成 RESTful API 文档。

### 6.1.8 RESTful API 实现

Spring MVC、Jersey、Play Framework 等主流的 Web 开发框架都支持 RESTful 的接口编写。这里我们以 Spring MVC 为例:

```
@RequestMapping(value = "/api/notes/{noteId}", method = RequestMethod.
GET, headers = "Accept=application/json")
@ResponseBody
public UserNote getUserNoteInfo(@PathVariable long noteId) {

    return ...;
}
```



此外，OAuth 的实现可以使用 Spring Security OAuth，其基于 Spring Security 实现了 OAuth 服务。不过，Spring Security OAuth 使用稍显复杂，完全可按照 OAuth 2.0 的流程使用 Spring MVC + Redis 进行实现。

## 6.2 远程过程调用——RPC

RPC，Remote Procedure Call，顾名思义就是远程过程调用，一般都有跨语言支持。大规模分布式应用中普遍使用 RPC 来做内部服务、模块之间的数据通信，还有助于解耦服务、系统的垂直拆分，使得系统可扩展性更强，并能够让 Java 程序员用与开发本地程序一样的语法与方式去开发分布式应用程序。

RPC 分为客户端（服务调用方）和服务端（服务提供方），都运行在自己的 JVM 中。客户端只需要引入要使用的接口，接口的实现和运行都在服务端。RPC 主要依赖的技术包括序列化、反序列化和数据传输协议，这是一种定义与实现相分离的设计，如图 6-3 所示。

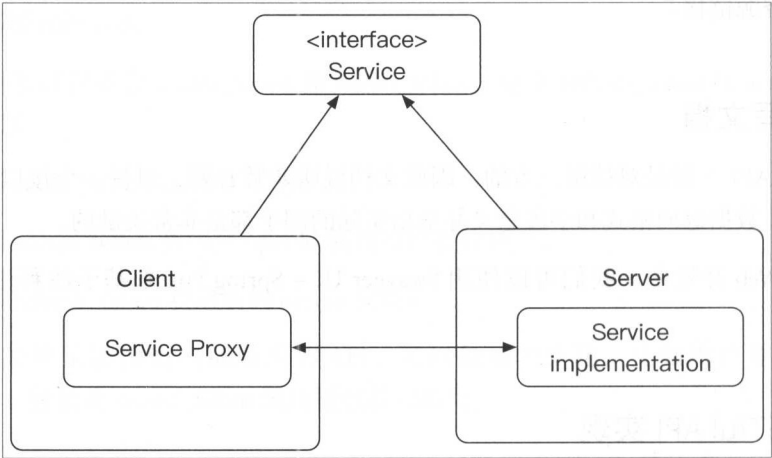


图 6-3

目前 Java 使用比较多的 RPC 方案主要有 RMI、Hessian、Dubbo 以及 Thrift 等。

这里需要提出的一点就是，这里的 RPC 主要指内部服务之间的调用，因此虽然 6.1 节的 RESTful 也可以用于内部服务间的调用，但其主要用途还在于为外部系统提供服务，因此本节没有将其包含在内。

## 6.2.1 JDK 自带的 RPC——RMI

RMI, Remote Method Invoke, 远程方法调用。是 Java 自带的远程方法调用工具, 其基于 TCP 连接, 可以使用任意端口, 不易跨网段调用, 不能穿越防火墙。但它是 Java 语言最开始时的设计, 后来很多框架的原理都基于 RMI。RMI 调用逻辑如图 6-4 所示。

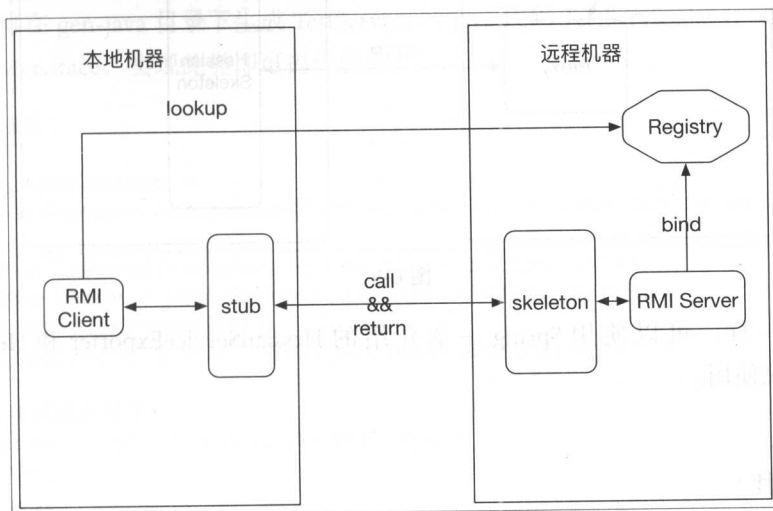


图 6-4

- **服务注册**：服务器端注册服务绑定到注册中心 Registry。
- **服务查找**：客户端根据服务名从注册中心查询要使用的接口获取引用。
- **服务调用**：stub 序列化调用参数并将其发送给 Skeleton，后者调用服务方法，并将结果序列化返回给 stub。

其序列化和反序列化使用的都是 JDK 自带的序列化机制。

这里服务注册管理中心在服务器端。其实可以完全独立出来作为一个单独的服务，其他的 RPC 框架很多都选择 ZooKeeper 充当此角色。

可以使用 Spring 一章介绍的 RmiServiceExporter 和 RmiProxyFactoryBean 来使用 RMI。

## 6.2.2 Hessian

Hessian 是一个基于 HTTP 协议的 RPC 方案, 其序列化机制是自己实现的, 负载均衡和容错需要依赖于 Web 容器 / 服务。其体系结构和 RMI 类似, 不过并没有注册中心 Registry 这一角色, 而是使用地址来显式调用。其中需要使用 HessianProxyFactory 根据配置的地址创建一个代理对象, 使用此代理对象去调用服务, 其调用逻辑如图 6-5 所示。

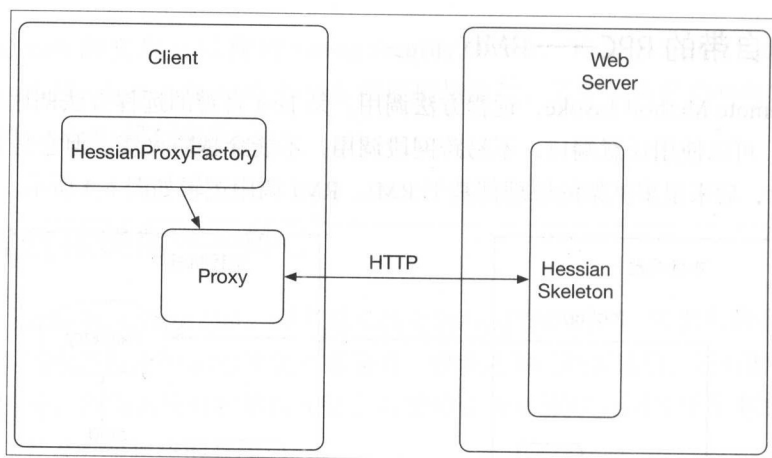


图 6-5

和 RMI 一样，可以使用 Spring 一章介绍的 `HessianServiceExporter` 和 `HessianProxyFactoryBean` 来使用。

### 6.2.3 Thrift

Thrift 是 Facebook 开源的 RPC 框架，现已进入 Apache 开源项目。其采用接口描述语言（IDL）定义 RPC 接口和数据类型，通过编译器生成不同语言的代码（支持 C++、Java、Python、Ruby 等），数据传输采用二进制格式，是自己实现序列化机制的，没有注册中心的概念，其调用逻辑如图 6-6 所示。

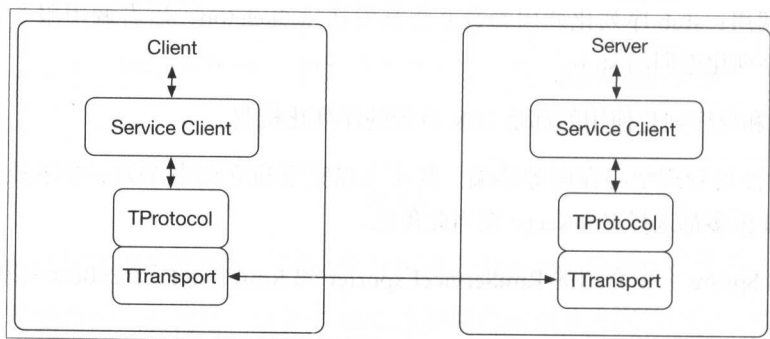


图 6-6

Thrift 的使用需要先编写接口的 IDL，然后使用它自带的工具生成代码：

```
namespace java me.rowkey.pje.datatrans.rpc.thrift
```

```
typedef i32 int
```

```
service TestService
```

```
{
    int add(1:int n1, 2:int n2),
}
```

// 代码生成

```
thrift --gen java TestService.thrift
```

以上即可在 gen-java 目录下生成 TestService 的 Java 代码 TestService.java, 其中的核心是接口 TestService.Iface, 实现此类即可提供服务。

服务提供方:

```
TProcessor tprocessor =
    new TestService.Processor<TestService.Iface>(new TestServiceImpl());

TServerSocket serverTransport = new TServerSocket(8088);
TServer.Args tArgs = new TServer.Args(serverTransport);
tArgs.processor(tprocessor);
tArgs.protocolFactory(new TBinaryProtocol.Factory());

// 简单的单线程服务模型
TServer server = new TSimpleServer(tArgs);
server.serve();
```

服务消费方:

```
TTransport transport = new TSocket("localhost", 8088, TIMEOUT);
TestService.Client testService =
    new TestService.Client(new TBinaryProtocol(transport));
transport.open();

int result = testService.add(1,2);
...
```

这里需要说明的一点就是, Thrift 提供了多种服务器模型、数据传输协议以及传输层供选择。

- 服务提供者的服务模型除了上面用的 TSimpleServer 简单单线程服务模型, 还有以下几个常用的模型。
  - TThreadPoolServer: 线程池服务模型, 使用标准的阻塞式 I/O, 预先创建一组线程处理请求。
  - TNonblockingServe: 非阻塞式 I/O。
  - THsHaServer: 半同步、半异步的服务器端模型。
- 数据传输协议除了上面例子使用的 BinaryProtocol 二进制格式, 还有下面这几种。
  - TCompactProtocol: 压缩格式。

- TJSONProtocol: JSON 格式。
- TSimpleJSONProtocol: 提供 JSON 只写协议,生成的文件很容易通过脚本语言解析。
- 传输层除了上面例子的 TServerSocket 和 TSocket, 还有下面这几种。
  - TFramedTransport: 以 frame 为单位进行传输,在非阻塞式服务中使用。
  - TFileTransport: 以文件形式进行传输。
  - THttpClient: 以 HTTP 协议的形式进行传输。

## 6.2.4 Dubbo

Dubbo 是阿里开源的服务治理框架。与前面介绍的几个 RPC 协议相比, Dubbo 不仅是一个 RPC 框架,还包含了服务治理方面的很多功能。

- 服务注册。
- 服务自动发现。
- 负载均衡。
- 集群容错。

这里仅针对 Dubbo 的 RPC 协议来介绍,其传输是基于 TCP 协议的,使用了高性能的 NIO 框架 Netty,序列化可以有多种选择,默认使用 Hessian 的序列化实现。Dubbo 默认使用 ZooKeeper 作为服务注册、管理中心。其实现逻辑如图 6-7 所示。

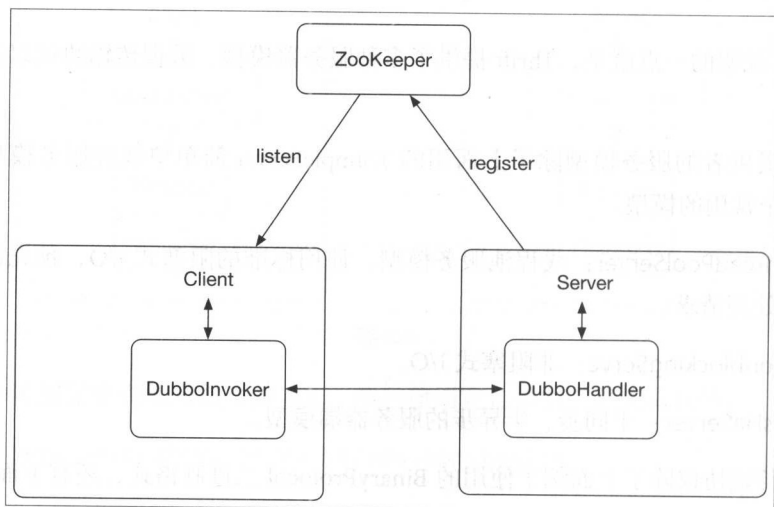


图 6-7

一个基于 Spring XML 配置的例子如下。

- 服务提供者 XML 配置。

```
<!-- 消费方应用名，用于计算依赖关系，不是匹配条件，不要与提供方一样 -->
<dubbo:application name="test_server"/>

<!-- 使用 ZooKeeper 注册中心暴露服务地址 -->
<dubbo:registry address="zookeeper://zk1.dmp.com:2181?backup=zk2.
dmp.com:2181,zk3.dmp.com:2181" file="${catalina.base}/logs/
eservice/dubbo.cache"/>

<dubbo:service path="emailService" interface="me.rowkey.pje.rpc.
test.service.IEmailService" ref="emailApiService" />
```

- 服务消费者 XML 配置。

```
<!-- 提供方应用信息，用于计算依赖关系 -->
<dubbo:application name="test_consumer"/>

<!-- 使用 ZooKeeper 注册中心 -->
<dubbo:registry address="zookeeper://zk1.dmp.com:2181?backup=zk2.
dmp.com:2181,zk3.dmp.com:2181" />

<dubbo:reference id="emailService" interface="me.rowkey.pje.rpc.
test.service.IEmailService"/>
```

在相关 Bean 中注入 emailService 即可使用。

## 6.2.5 数据的序列化机制

序列化是 RPC 的一个很关键的地方，序列化、反序列化的速度、尺寸大小都关系着 RPC 的性能。包括上面提到的几个序列化协议，现在使用较普遍的 Java 序列化协议有以下几种。

### 1) Java Serialiazer

JDK 自带的序列化机制，使用起来比较方便。但是其是对对象结构到内容的完全描述，包含所有的信息，因此速度较慢，占用空间也比较大，且只支持 Java 语言。一般不推荐使用。

需要注意的是，字段 serialVersionUID 的作用是为了在序列化时保持版本的兼容性，即在版本升级时反序列化仍保持对象的唯一性。否则如果你在序列化后更改 / 删除了类的字段，那么在反序列化时就会抛出异常。而如果设置了此字段的值，那么会将不一样的 field 以 type 的预设值填充。如下：

// 序列化

```
ByteArrayOutputStream bout = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(bout);
out.writeObject(obj);
byte[] bytes = bout.toByteArray();
```

// 反序列化

```
ObjectInputStream bin = new ObjectInputStream(new
ByteArrayInputStream(bytes));
bin.readObject();
```

## 2) Hessian

底层是基于 list 和 HashMap 实现的，着重于数据，附带简单的类型信息的方法，支持多种语言，兼容性比较好，与 JDK 序列化相比高效且占用空间较小；但其在序列化的类有父类的时候，如果有字段相同，父类的值会覆盖子类的值，因此使用 Hessian 时一定要注意子类和父类不能有同名字段。

需要注意的一点是，Hessian 的实现里有 v1 和 v2 两种版本的协议支持，并不兼容，推荐使用 Hessian 2 相关的类。

与后来出现的其他二进制序列化工具相比，Hessian 的速度和空间都不是优势。

// 序列化

```
ByteArrayOutputStream os = new ByteArrayOutputStream();
Hessian2Output out = new Hessian2Output(os);
out.startMessage();
TestUser user = new TestUser();
out.writeObject(user);
out.completeMessage();
out.flush();
byte[] bytes = os.toByteArray();
out.close();
os.close();
```

// 反序列化

```
ByteArrayInputStream ins = new ByteArrayInputStream(bytes);
Hessian2Input input = new Hessian2Input(ins);
input.startMessage();
TestUser newUser = (TestUser)input.readObject();
input.completeMessage();
input.close();
ins.close();
```

## 3) MsgPack

MsgPack 是一个非常高效的对象序列化库，支持多种语言，有点像 JSON，但是非常快，且占用空间也较小，号称比 Protobuf 还要快 4 倍。

使用 MsgPack 需要在序列化的类上加 @Message 注解。为了保证序列化向后兼容,新增的属性需要加在类的最后面,且要加 @Optional 注解,否则反序列化会报错。

此外,MsgPack 提供了动态类型的功能,通过接口 Value 来实现动态类型,首先将字节数组序列化为 Value 类型的对象,然后用 converter 转化为本身的类型。

MsgPack 不足的一点就是其序列化和反序列化都非常消耗资源。

```
//TestUser.java
@Message
public class TestUser{
    private String name;
    private String mobile;
    ...
}

TestUser user = new TestUser();
MessagePack messagePack = new MessagePack();

// 序列化
byte[] bs = messagePack.write(user);

// 反序列化
user = messagePack.read(bs, TestUser.class);
```

#### 4) Kryo

Kryo 是一个快速高效的 Java 对象图形序列化框架,使用简单、速度快、序列化后体积小。其实现代码非常简单,远远小于 MsgPack。但其文档较少,跨语言支持也较差,适用于 Java 语言。目前 Kryo 的版本到了 4.x,对于 2.x 之前版本的很多问题都做了修复。

```
Kryo kryo = new Kryo();

// 序列化
ByteArrayOutputStream os = new ByteArrayOutputStream();
Output output = new Output(os);
TestUser user = new TestUser();
kryo.writeObject(output, user);
output.close();
byte[] bytes = os.toByteArray();

// 反序列化
Input input = new Input(new ByteArrayInputStream(bytes));
TestUser newUser = kryo.readObject(input, TestUser.class);
input.close();
```



### 5) Thrift

上面介绍的 Thrift RPC 框架其内部的序列化机制可以单独使用，主要是对 TBinaryProtocol 的使用。和接口的生成方式类似，需要先定义 IDL，再使用 Thrift 生成。其序列化性能比较高，空间占用也比较小。但其设计目标并非是单独作为序列化框架使用的，一般都是整体作为 RPC 框架使用的。

定义 IDL:

```
//TestUser.thrift
namespace java me.rowkey.pje.datatrans.rpc.thrift

struct TestUser {
    1: required string name
    2: required string mobile
}
```

```
thrift --gen java TestUser.thrift
```

使用生成的 TestUser 类做序列化和反序列化:

```
TestUser user = new TestUser(); // 由 Thrift 代码生成引擎生成
```

// 序列化

```
ByteArrayOutputStream bos = new ByteArrayOutputStream();
user.write(new TBinaryProtocol(new TIOStreamTransport(bos)));
byte[] result = bos.toByteArray();
bos.close();
```

// 反序列化

```
ByteArrayInputStream bis = new ByteArrayInputStream(result);
TestUser user = new TestUser();
user.read(new TBinaryProtocol(new TIOStreamTransport(bis)));
bis.close();
```

需要注意的是，由于 Thrift 序列化时，丢弃了部分信息，故使用 ID+Type 来做标识，因此对新增的字段属性采用 ID 递增的方式标识并以 Optional 修饰来添加，这样才能做到向后兼容。

### 6) Protobuf

Protobuf 是 Google 开源的序列化框架，是 Google 公司内部的混合语言数据标准，用于 RPC 系统和持续数据存储系统，其非常轻便高效，具有很好的可扩展性，也具有良好的向后兼容性和向前兼容性。与上述的几种序列化框架对比，序列化数据紧凑、速度快、空间占用小、资源消耗较低、使用简单，但其缺点在于需要静态编译生成代码、可读性差、缺乏自描述、向后兼容有一定的约束限制。

这里需要注意，目前 Protobuf 的版本到了 3.x，比 2.x 支持更多语言但更简洁。其去掉了一些复杂的语法和特性，更强调约定而弱化语法。因此，如果是首次使用就直接使用 3.x 版本。本书也是针对 Protobuf 3 来介绍的。

首先需要编写 .proto 文件，并使用 Protobuf 代码生成引擎生成 Java 代码：

```
//TestUser.proto
syntax = "proto3";
option java_package = "me.rowkey.pje.datatrans.rpc.proto";
option java_outer_classname = "TestUserProto";
message TestUser
{
    string name=1;
    string mobile=2;
}

protoc --java_out=./ TestUser.proto
```

即生成 TestUserProto.java，使用此类即可完成序列化和反序列化：

```
// 序列化
TestUserProto.TestUser testUser =
    TestUserProto.TestUser.newBuilder()
        .setMobile("xxx")
        .setName("xxx")
        .build();

byte[] bytes = testUser.toByteArray();

// 反序列化
testUser = TestUserProto.TestUser.parseFrom(bytes);
```

综上，对以上几个序列化框架做对比如表 6-1 所示。

表 6-1

	优点	缺点
Java	JDK 自带实现，包含对象的所有信息	速度较慢，占用空间也比较大，只支持 Java 语言
Hessian	支持语言比较多，兼容性较好	较慢
MsgPack	使用简单、速度快、体积小	兼容性较差，耗资源
Kryo	速度快，序列化后体积小	跨语言支持较差，文档较少
Thrift	高效	需要静态编译；是 Thrift 内部序列化机制，很难和其他传输层协议共同使用
Protobuf	速度快	需要静态编译

在兼顾使用简单、速度快、体积小且主要使用在 Java 开发的场景下, Kryo 是比较好的方案; 如果对占用空间、性能等有特别的要求, 那么 Protobuf 则是更好的选择。此外, JSON 其实也是一种序列化方式, 如果比较关注阅读性, 那么 JSON 是更好的选择。

### 6.2.6 使用提示

面对这些 RPC 框架, 选择的时候应该从以下几方面进行选择。

- **是否允许代码侵入:** 即需要依赖相应的代码生成器生成代码, 比如 Thrift。
- **是否需要长连接获取高性能:** 如果性能需求比较高, 那么果断选取基于 TCP 的 Thrift、Dubbo。
- **是否需要跨网段、跨防火墙:** 这种情况一般需要选择基于 HTTP 协议的 Hessian 和 Thrift 的 HTTP Transport。

此外, 除了上述框架外, Google 推出的基于 HTTP 2.0 的 gRPC 框架也开始得到了应用, 其序列化协议基于 Protobuf, 网络框架使用了 Netty 4。但其需要生成代码, 可扩展性也比较差。

## 6.3 消息中间件

消息中间件, 也可以叫作中央消息队列(区别于本地消息队列), 是一种独立的队列系统。消息中间件经常用来解决内部服务之间的异步调用问题。调用方把请求放到队列中即可返回, 然后等待服务提供方去队列中获取请求进行处理, 之后通过回调等机制把结果返回给调用方。

异步调用就是消息中间件一个非常常见的应用场景。此外, 常用的消息队列的应用场景还有以下几个。

- **解耦:** 一个业务的非核心流程需要依赖其他系统, 但结果并不重要, 有通知即可。
- **最终一致性:** 指的是两个系统的状态保持一致, 可以有一定的延迟, 只要最终达到一致性即可。经常用在解决分布式事务上。
- **广播:** 这是消息队列最基本的功能。生产者只需要发布消息, 订阅者都会收到消息。
- **错峰与流控:** 当上下游系统处理能力不同的时候就需要类似消息队列的方式作为缓冲区来隔开两个系统。

其实，这里消息队列的概念和操作系统中进程间通信方式的消息队列的概念本质是相同的。消息队列有生产者和消费者两种角色，前者生产消息，后者消费消息。

对于生产者生产消息、消费者消费消息，大体有两种消息模型：队列和发布 / 订阅。

- 在队列模型中，一个由消费者组成的池从服务器读取消息，每一个消息都可以达到其中的某一个消费者，如图 6-8 所示。

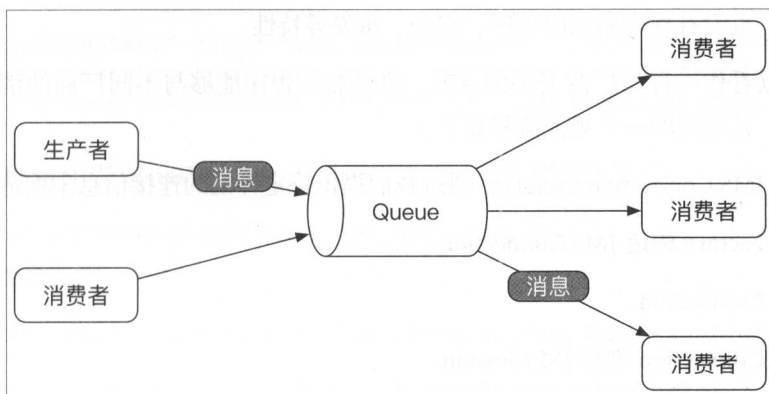


图 6-8

- 在发布 / 订阅模型中，消息被广播到所有消费者中，所有订阅了某个 Topic（主题）的消费者都能够拿到消息，如图 6-9 所示。

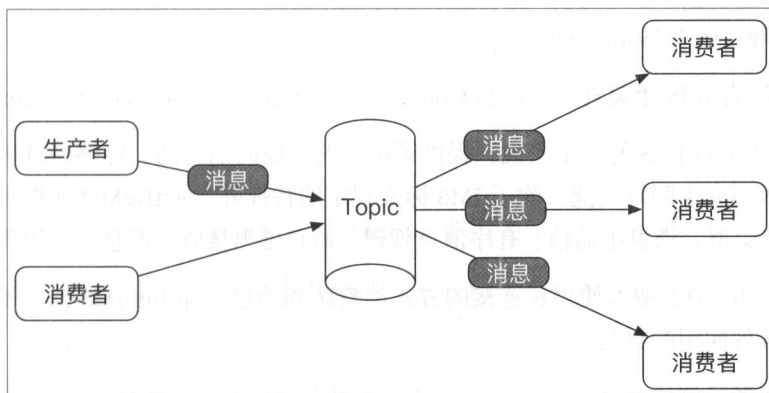


图 6-9

目前主流的消息中间件有以下几个。

- ActiveMQ。
- RabbitMQ。
- Kafka。

### 6.3.1 简单消息中间件——ActiveMQ

理解 ActiveMQ，需要先讲述 JMS 规范。

JMS (Java Message Service) 是一个简单的消息中间件规范，专用于 Java EE。它主要做了接口上的规范 (定义了 API 接口) 以及消息传输模型和消息类型的规范。但其并没有对这些给予实现，也完全没有给出服务器端的架构，甚至可以不使用服务器直接在客户端之间传输消息；也没有规定消息的顺序、安全、重发等特性。

JMS 可以看作一种与厂商无关的 API，使得 Java 程序能够与不同厂商的消息组件很好地进行通信。其定义的一个规范流程如下。

- 获得 JMS Connection Factory，通过我们提供特定环境的连接信息来构造 Factory。
- 使用 Factory 构造 JMS Connection。
- 启动 Connection。
- 通过 Connection 创建 JMS Session。
- 指定 JMS Destination。
- 创建 JMS Producer 或者创建 JMS Message 并提供 Destination。
- 创建 JMS Consumer 或注册 JMS Message Listener。
- 发送和接收 JMS Message。
- 关闭所有 JMS 相关资源，包括 Connection、Session、Producer、Consumer 等。

ActiveMQ 是对 JMS 的一个实现，提供了标准的、面向消息的、能够跨越多语言和多系统的应用集成消息中间件功能。除了 JMS 标准规定的特性外，ActiveMQ 还提供了诸如消息持久化、主从集群、消息组通信、有序消息管理、消息延迟接收、消息优先级等附加特性。

在 ActiveMQ 中消费者使用长连接的方式消费队列消息。Spring JMS 提供了对符合 JMS 规范的消息队列使用的封装。

消息生产者示例如下：

```
<bean id="amqConnectionFactory" class="org.apache.activemq.
ActiveMQConnectionFactory">
    <constructor-arg index="0" value="tcp://localhost:61616"/>
</bean>

<!-- 定义连接工厂 -->
<bean id="connectionFactory" class="org.springframework.jms.connection.
CachingConnectionFactory">
```

```

    <constructor-arg ref="amqConnectionFactory" />
</bean>

<!-- 配置目的队列 -->
<bean id="defaultDestination" class="org.apache.activemq.command.
ActiveMQQueue">
    <constructor-arg index="0" value="test_jms_ueue" />
</bean>

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="defaultDestination" ref="defaultDestination" />
</bean>

```

在相关 Bean 中引入 jmsTemplate 即可调用相关方法实现消息发送:

```
jmsTemplate.convertAndSend("test_jms_queue",message);
```

消息消费者示例:

```

<bean class="org.springframework.jms.listener.SimpleMessageListenerContai
ner">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destinationName" value="test_jms_queue" />
    <property name="messageListener" ref="activeMQListener" />
</bean>

@Component("activeMQListener")
public class ActiveMQMessageListener implements javax.jms.MessageListener.
MessageListener {
    @Override
    public void onMessage(Message message) {
        ...// 消息处理逻辑
    }
}

```

实现 javax.jms.MessageListener.MessageListener 即可处理收到的消息。

由于 ActiveMQ 性能较差,在大规模的互联网应用中并不推荐使用它。

### 6.3.2 通用消息中间件——RabbitMQ

RabbitMQ 基于 AMQP 协议,是一个完备的消息中间件。

#### AMQP

AMQP, The Advanced Message Queuing Protoco, 是一个比较复杂的消息中间件协议。它是一个和语言无关的,在金融行业使用的新兴消息中间件,是一个异步消息传递所使用的应用层协议规范。其现在的目标是为通用消息队列架构提供通用构建工具。

与 JMS 相比, AMQP 并不是 API, AMQP 客户端能够无视消息的来源任意发送和接收信息。AMQP 提供了一个模型, 旨在统一消息模式, 包括队列、发布/订阅、事务等, 还包括路由、扩展性等额外特性, 能够让客户端和消息中间件之间进行消息通信及消息确认, 如图 6-10 所示。

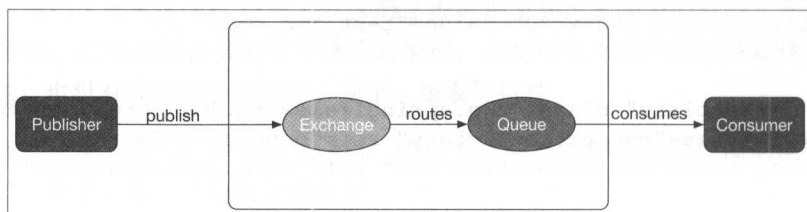


图 6-10

如图 6-10 所示, AMQP 工作模式可以描述为: 消息首先被发布到信箱 (Exchange), 然后信箱会把消息复制分发到应用了规则 (即绑定 Binding) 的队列。AMQP 消息中间件既可以将消息分发到订阅了某些队列的消费者, 也可以让消费者自己根据需从队列中拉取消息。其中:

- 信箱 Exchange、队列 Queue 和绑定 Binding 被统称为 AMQP 实体。
  - **信箱**: 用于消息发送的实体, 可以将一条消息路由到零个或多个队列。具有名称、持久性、自动删除等属性, 这里的自动删除指的是信箱会在所有队列都不再使用它的时候被删除。
  - **队列**: 和其他消息队列软件很类似, 会存储供应用程序消费的消息。队列会和信箱共享一些属性, 但也有一些额外的属性, 包括名称、持久性、连接的专一性、自动删除、消息的 TTL 等。这里的连接专一性指的是队列只被一个连接使用, 并在连接断开时删除; 自动删除指的是当最后一个消费者取消订阅后, 队列会被删除。
  - **绑定**: 是信箱用于路由消息到队列的一些规则, 可以认为绑定就是连接信箱和队列的路径。
- 在发布消息时, 发布者可以定义各种消息属性。其中的一些属性可能会被消息中间件使用, 余下的则由接收消息的应用程序使用。
- 当消息传递到消费者时, 消费者会给消息中间件发送确认通知, 可以自动回复或者选择在需要的时候回复。当使用消息确认机制时, 只有在消息中间件接收到来自消费者的确认时才会将消息从队列中移除。

- 消息的负载均衡是在消费者之间进行的，即当多个消费者监听同一个 Queue 时，使用 Round Robin 策略将消息只传输给一个消费者。
- AMQP 对于消费者的实现包括 Push 和 Pull 两种方式。

此外，需要注意 AMQP 是一个抽象协议，不负责处理具体的数据。它的实体和路由策略都可以根据需要定义并可以在不需要的时候删除这些 AMQP 实体。

## RabbitMQ 介绍

RabbitMQ 是实现了 AMQP 的开源消息队列，服务器端用 Erlang 语言编写，支持多种客户端。从生产者接收消息并传递给消费者。在这个过程中，根据规则进行路由、缓存以及持久化。消费者通过 Push 方式获取消息，即队列里有消息就会推送给消费者。其架构基于 AMQP 模型，如图 6-11 所示。

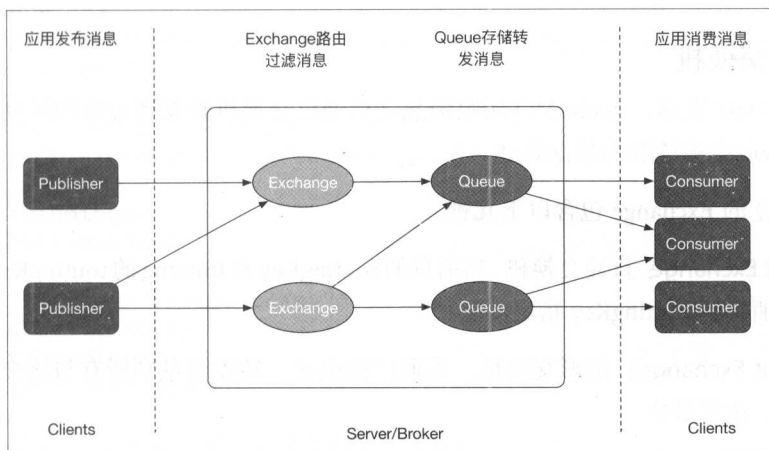


图 6-11

比起 AMQP 模型，RabbitMQ 增加了以下几个概念。

- Broker：消息队列服务器实体，一个 RabbitMQ 实例就是一个 Broker。
- VHost：虚拟主机，一个 Broker 里可以开设多个 VHost，用作不同用户的权限分离。
- Channel：消息通道，在客户端的每个连接里，可建立多个 Channel，每个 Channel 代表一个会话任务。

RabbitMQ 的使用过程如下。

- 客户端连接到消息队列服务器，打开一个 Channel。
- 客户端声明一个 Exchange，并设置相关属性。



- 客户端声明一个 Queue，并设置相关属性。
- 客户端使用 routingKey，在 Exchange 和 Queue 之间建立绑定关系。
- 客户端投递消息到 Exchange。
- Exchange 接收消息后，就根据消息的 key 和已经设置的 Binding 进行消息路由，将消息投递到一个或多个队列里。

RabbitMQ 具有较高的可用性、稳定性以及可靠性，具备了一个成熟的消息队列应该具有的特性。其适用于可靠性要求极高、对消息有事务要求的业务场景。

需要注意的是，RabbitMQ 也只能保证 At Lease Once（消息至少被消费一次）的消费，无法从自身进行消息去重。如果要达到 Exactly Once（消息会且仅会被消费一次），应该由业务去做去重处理或者业务消息本身就具有幂等性。

## RabbitMQ 交换机

消息由 Client 发送，RabbitMQ 接收消息之后通过交换机转发到对应的队列上。消费者会从队列中获取未被读取的数据处理。

RabbitMQ 的 Exchange 包含以下几种。

- Direct Exchange 直连交换机。当消息的 routingKey 和 Binding 的 routingKey 直接匹配，转发消息到 routingKey 指定的队列。
- Fanout Exchange：扇形交换机。采取广播模式，转发消息到所有与该交换机绑定的队列，速度最快。
- Topic Exchange：主题交换机。使用匹配模式按规则转发消息，即判断消息的 routingKey 和 Binding 的 routingKey 是否符合通配符匹配。这是最灵活的交换机。
- Headers Exchange：头部交换机。如果消息的头部信息和 Binding 的参数表匹配的话，消息将会路由到该队列。

## 消息持久化

RabbitMQ 支持消息的持久化，包括如下 3 个方面。

- Exchange 持久化，在声明时指定 `durable ≥ 1`。
- Queue 持久化，在声明时指定 `durable ≥ 1`。
- 消息持久化，在投递时指定 `delivery_mode ≥ 2`。

这里如果 Exchange 和 Queue 都是持久化的, 那么它们之间的 Binding 也是持久化的。如果 Exchange 和 Queue 两者之间有一个持久化, 一个非持久化, 则就不允许建立绑定。

## 使用

Spring Rabbit 和 Spring AMQP 对 RabbitMQ 的使用做了封装。

生产者配置:

```
<!-- 配置 connection-factory, 指定连接 Rabbit Server 参数 -->
<rabbit:connection-factory id="rabbitConnectionFactory"
    username="xx"
    password="xx"
    host="localhost"
    port="5672" />

<!-- 定义 queue -->
<rabbit:queue name="testQueue" durable="true" auto-delete="false"
    exclusive="false" />

<!-- 定义 direct exchange, 绑定 queue -->
<rabbit:direct-exchange name="testExchange" durable="true" auto-
    delete="false">
    <rabbit:bindings>
        <rabbit:binding
            queue="testQueue"
            key="testQueueKey">
        </rabbit:binding>
    </rabbit:bindings>
</rabbit:direct-exchange>

<!-- 定义 Rabbit template 用于数据的接收和发送 -->
<rabbit:template id="amqpTemplate" connection-
    factory="rabbitConnectionFactory"
    exchange="testExchange" />

<!-- 通过指定下面的 admin 信息, 当前 Producer 中的 Exchange 和 Queue 会在 RabbitMQ 服
    务器上自动生成 -->
<rabbit:admin connection-factory="rabbitConnectionFactory" />
```

这样引入 amqpTemplate 即可调用相关方法发送消息:

```
amqpTemplate.convertAndSend("testQueueKey", message);
```

消费者配置:

```
<!-- 定义 queue -->
<rabbit:queue name="testQueue" durable="true" auto-delete="false"
    exclusive="false" />
```

```

<!-- 定义 direct exchange, 绑定 queue -->
<rabbit:direct-exchange name="testChange" durable="true" auto-
delete="false">
    <rabbit:bindings>
        <rabbit:binding
            queue="testQueue"
            key="testQueueKey">
        </rabbit:binding>
    </rabbit:bindings>
</rabbit:direct-exchange>

<rabbit:listener-container
    connection-factory="rabbitConnectionFactory">
    <rabbit:listener queues="testQueue"
        ref="amqpListener"/>
</rabbit:listener-container>

@Component("amqpListener")
public class AmqpMessageListener implements org.springframework.amqp.
core.MessageListener {
    @Override
    public void onMessage(Message message) {

    }
}

```

如上, 实现 `org.springframework.amqp.core.MessageListener` 即可接收消息进行处理。

### 6.3.3 日志消息中间件——Kafka

Kafka 是由 LinkedIn 公司开发的一个分布式的消息系统, 后来成为 Apache 的开源项目, 其使用 Scala 编写, 以可水平扩展和高吞吐率而被广泛使用。目前越来越多的开源分布式处理系统如 Apache Storm、Spark 等都支持与 Kafka 集成。

Kafka 基于文件 append, 以顺序读 / 写的方式来写入 / 读取文件, 性能 (吞吐量、TPS) 是非常高的, 也支持多订阅者, 当失败时能自动平衡消费者。但由于 Kafka 设计的初衷就是处理日志, 因此是允许消息丢失的, 对事务的支持也不好, 并不能保证消息的顺序。此外, Kafka 中消费者是通过 Pull 的方式获取消息的, 因此在实时性上不如 RabbitMQ, 但是其使用 Zero-Copy 技术, 保证了一定的 Pull 性能。

综上, Kafka 适用于海量消息场景, 允许极端情况下少量丢失数据, 如日志。

本书使用的 Kafka 为 0.9.0 版本。

## 关键概念和架构

几个概念如下。

- Topic: Kafka 将消息以 category 的方式保存在一起, 称为 Topic。
- Producer: 向 Topic 产生消息的进程被称为 Producer。
- Consumer: 处理 Topic 上的消息的进程被称为 Consumer。
- ConsumerGroup: 每个 Consumer 属于一个特定的 Consumer Group, 一条消息可以发送到多个不同的 Consumer Group, 但是一个 Consumer Group 中只能有一个 Consumer 消费该消息。
- Broker: Kafka 集群由一个或者多个 Server 组成, 称为 Broker。
- Partition: 分区, 是一个物理上的概念, 一个 Topic 可以分为多个 Partition, 每个 Partition 内部是消息有序的。

Kafka 的架构如图 6-12 所示。

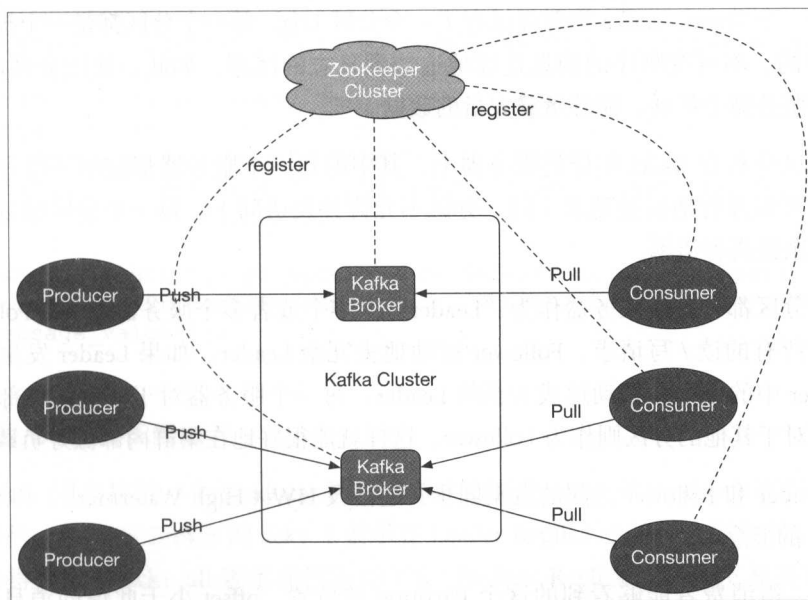


图 6-12

消息的发送和接收流程如下。

- 通过 ZooKeeper 管理集群配置、选举 Leader; 在 Consumer Group 变化时进行 rebalance。

- 生产者以 Push 方式向所选择的 Topic 发布消息并负责选择哪一个消息被指定到 Topic 的哪一个 Partition 中。这个可以通过 round-robin 简单地做负载均衡或者按照一些语义分区机制（例如基于消息中的一些 key）来做。
- 消费者使用 Pull 方式订阅消息。基于 Consumer Group 可以实现两种消费模式。
  - 所有的消费者实例都在同一个 Consumer Group 中，那么就类似于传统的队列，同一 Group 中的 Consumer 对同一消息仅仅能获取一次。
  - 每一个消费者实例都在不同的 Consumer Group 中，那么就类似于发布 / 订阅模型，所有消息被广播到所有消费者。

需要提到的一点是，Kafka 的所有消息都是顺序 append 到日志文件中的，每条消息在文件中的位置被称为 offset（偏移量），offset 为一个 long 型的数字，它唯一标记一条消息。因此，Consumer 对 offset 的管理是读取消息的关键点之一。

## 分布式

对于每一个 Topic，Kafka 集群都保存了一个分区 Log，每一个分区都是一个提交日志，一系列有序的、不可变顺序的消息连续地追加到日志的尾部。如此，使用分区使得 Kafka 可以在单个服务器上扩展，能够承载大量的数据。

这些分区分布在 Kafka 集群的服务器上，其中的每一个服务器都控制一组分区上的数据和请求，可以并行地接受请求（同一分区不允许并发访问）。每一个分区通过一定数量的服务器冗余提高容错率。

每一个分区都有一个服务器作为“Leader”，零个或者多个服务器作为“Follower”。Leader 控制所有的读 / 写请求，Follower 被动地去冗余 Leader。如果 Leader 发生了故障，那么 Follower 中的一个会自动地成为新的 Leader。每一个服务器对于其中的一部分分区作为 Leader，对于其他的分区则作为 Follower，这样就能很好地在集群内部做好负载均衡。

对于 Leader 和 Follower 之间的数据同步，还涉及 HW（High Watermark）和 LEO（Log End Offset）的概念。

- HW：指消费者能够看到的这个 Partition 的位置，offset 小于此值的消息被认为是 commit 的，可以供消费者消费。
- LEO：每一个 Replica 的 Log 中最后一条消息的 offset。

Follower 拉取 Leader 数据，当数据完全同步时则阻塞。当有新的数据到来，Leader 会解锁阻塞的 Follower 通知它们新的消息，Follower 开始同步消息并更新自己的 LEO。Leader 选取所有 ISR（In-Sync Replica，指 alive 和追赶 Leader 的 Replica）中最小的 LEO 作为自己

的 HW，消费者最多只能消费 HW 所在的位置。由此可见，Kafka 的数据复制机制既不是完全的同步复制，也不是单纯的异步复制，其使用 ISR 的这种方式能够很好地均衡吞吐率和数据的安全性。

由于分区的原因 Kafka 无法保障全局消息有序，但通过指定分区到一个 Consumer Group 中的 Consumer，这样每一个分区只被这个 Group 中的一个 Consumer 消费，能够保证一个分区中的消息顺序。当然，使用一个只有一个分区的 Topic 能够保证消息全局有序。

## 使用

生产者示例：

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "all");
props.put("request.timeout.ms", "10000");
props.put("retries", 10);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serialization.
StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.
StringSerializer");
props.put("compression.codec", "none");
props.put("partitioner.class", "org.apache.kafka.clients.producer.
internals.DefaultPartitioner");

Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<String, String>("test_topic", "message_
key", "message_value"));
...
producer.close();
```

其中的几个关键配置如下。

- **acks**: 用来控制一个 Producer 请求怎样才算完成。0 表示 Producer 直接返回，不会等待一个来自 Broker 的 ack；1 表示在 Leader Replica 已经接收到数据后，Producer 会得到一个 ack；all 表示在所有的 ISR（In-Sync Replica，指 alive 和追赶 Leader 的 Replica）都接收到数据后，Producer 才得到一个 ack。
- **batch.size**: 批量发送消息（同一个分区）时的一个 batch 的大小，单位 byte。过小的值会使得发送频繁，降低吞吐量；过大的值则会浪费内存；设置为 0 则不进行批量发送。

- `linger.ms`: 消息记录缓冲的时间, 即延时发送消息记录等待其他消息记录一起作为一个 batch。当满足 `batch.size` 时, 会忽略此值直接发送, 否则延长此时间再发送。
- `buffer.memory`: 用于缓冲消息记录的内存大小, 单位 byte。
- `key.serializer`: 消息的 key 的序列化类。
- `value.serializer`: 消息的 value 的序列化类。
- `compression.codec`: Producer 的数据压缩方式, 包括 None、GZIP、Snappy 和 LZ4 这 4 种。
- `partitioner.class`: 用来把消息分到各个 Partition 中的实现类, 默认对 key 进行 hash。

这里需要注意的是, 此处使用的是 `kafka-clients` 库中的 `KafkaProducer`, 是纯 Java 的实现。它是在 Kafka 0.8.2 版本之后被引入的, 与之前 Scala 版本的 Kafka 库中的 Producer 配置参数有不少区别。

消费者的 API 分为以下两种。

### 1) High Level

这种级别的 API 封装了很多底层细节, 使用 ZooKeeper 保存 offset 信息:

```
Properties props = new Properties();
props.put("zookeeper.connect", "zk1.dmp.com:2181,zk2.dmp.com:2181,zk3.dmp.com:2181");
props.put("zookeeper.session.timeout.ms", "3000");
props.put("zookeeper.sync.time.ms", "200");
props.put("group.id", "test_group");
props.put("auto.commit.interval.ms", "600");

String topic = "test_topic";
ConsumerConnector connector = Consumer.createJavaConsumerConnector(new ConsumerConfig(props));
Map<String, Integer> topics = new HashMap<String, Integer>();
int partitionNum = 3; // 分区数目
topics.put(topic, partitionNum);
Map<String, List<KafkaStream<byte[], byte[]>>> streams = connector.createMessageStreams(topics);
List<KafkaStream<byte[], byte[]>> partitions = streams.get(topic);
Executor threadPool = Executors.newFixedThreadPool(partitionNum);
for (final KafkaStream<byte[], byte[]> partition : partitions) {
    threadPool.execute(
        new Runnable() {
            @Override
            public void run() {
```

```

        ConsumerIterator<byte[], byte[]> it = partition.iterator();
        while (it.hasNext()) {
            MessageAndMetadata<byte[], byte[]> item = it.next();
            byte[] messageBody = item.message();
        }
    }
});
}

```

## 2) Low Level

Kafka 的 low-level 接口主要是对 SimpleConsumer 的使用，使用场景如下。

- 读取一个消息多次。
- 在一个进程中仅消费某一个 Topic 中几个 Partition 的数据。
- 管理事务以确保一个消息处理且仅被处理一次。

使用这个接口需要注意以下几点。

- 在应用中必须追踪记录 offset 以确保能够确定上次消费到的位置。
- 必须设置哪一个 Broker 是要操作的 Topic 和 Partition 的 Leader。
- 必须自己控制 Broker 的 Leader 的改变。

使用步骤如下。

- 找出一个 active 状态的 Broker 并且找出哪一个 Broker 是哪些 Topic 和 Partition 的 Leader，必须知道读哪个 Topic 的哪个 Partition。
- 找到负责该 Partition 的 Broker Leader，从而找到存有该 Partition 副本的那个 Broker。
- 自己去写 Request 并 fetch 数据。
- 获取数据。
- 需要识别和处理 Broker Leader 的改变。

Kafka 0.9.0 之后的 kafka-clients 提供了新的 KafkaConsumer 实现，不再区分 Hight Level 和 Low Level:

```

Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test_group");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("session.timeout.ms", "30000");

```



```

props.put("key.deserializer", "org.apache.kafka.common.serialization.
StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.
StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("test_topic"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records){
        ...
    }
}

```

如果需要自己手动控制 offset 的保存，可以将 `enable.auto.commit` 设置为 `false`，在相应的地方对 `consumer.commitSync()` 进行提交。

如果只需要消费某几个 Partition 的数据：

```

String topic = "test_topic";
TopicPartition partition0 = new TopicPartition(topic, 0);
TopicPartition partition1 = new TopicPartition(topic, 1);
consumer.assign(Arrays.asList(partition0, partition1));
...

```

如果需要自定义 offset 存储实现和对 offset 的控制，

- 将 `enable.auto.commit` 设置为 `false`。
- 获取 `ConsumerRecord` 的 offset 并保存。
- 使用 `Consumer` 的 `seek(TopicPartition, long)` 设置 offset。

此外，在开发 `Consumer` 程序时还有以下几个注意点。

1) 同一 `Consumer Group` 消费过的数据无法再次消费。如果想要再次消费数据，要么换另一个 `groupId`，要么使用镜像或者使用 `Low Level API` 或者新的 `Consumer API` 去设置 `Partition` 和 `offset`。此外，`Kafka` 本身也提供了工具来粗略地重新设置 `offset`：

```

./kafka-run-class.sh kafka.tools.UpdateOffsetsInZK earliest config/
consumer.properties page_visits

```

参数解释如下。

- `[earliest | latest]`，表示将 `offset` 设置到哪里。
- `consumer.properties`，这里是配置文件的路径。
- `Topic` 名，这里是 `page_visits`。

2) Kafka 0.8.2 版本引入了 native offset storage, 将 offset 管理从 ZooKeeper 移出, 并且可以做到水平扩展, 相比之前用 ZooKeeper 存储 offset, 避免了对 ZooKeeper 的频繁写入(低效操作)。

3) 在上面 Consumer 的 High Level API 中, Consumer 中的 Stream 指的是来自一个或多个服务器上的一个或者多个 Partition 的消息。每一个 Stream 都对应一个单线程处理。因此, Client 能够设置满足自己需求的 Stream 数目。总之, 一个 Stream 也许代表了多个服务器 Partition 的消息的聚合, 但是每一个 Partition 都只能对应一个 Stream。

4) Consumer 和 Partition 的数目需要配合设置。

- 如果 Consumer 比分区多, 则是浪费, 因为 Kafka 的设计在一个分区上是不允许并发的, 所以 Consumer 数不要大于分区数。
- 如果 Consumer 比分区少, 一个 Consumer 会对应多个分区, 需要合理分配 Consumer 数和 Partition 数, 否则会导致 Partition 里面的数据被取得不均匀。
- 如果 Consumer 从多个 Partition 读到数据, 不保证消息有序。
- 增减 Consumer、Broker、分区会导致 rebalance, 所以 rebalance 后 Consumer 对应的分区会发生变化。

综上, 在负载低的情况下可以每个线程消费多个 Partition。但在负载高的情况下, Consumer 线程数最好和 Partition 数量保持一致。如果还是消费不过来, 应该再开 Consumer 进程, 进程内线程数同样和分区数一致。

这里, 由于 Kafka 是对磁盘进行读/写, 因此可以考虑优化系统的 Page Cache 来提升其读/写性能。

还需要提到一点, 由于 Kafka 是使用 Scala 语言编写的, 因此阿里巴巴基于其原理(后来有了自己的设计)使用 Java 语言实现了 MetaQ 并开源, 现在已经变成 RocketMQ, 可以作为 Kafka 的替代品。

### 6.3.4 本地消息队列

区别于消息中间件, 本地消息队列指的是 JVM 内的队列实现, 常用的包括下面几个。

#### 1) LinkedBlockingQueue

阻塞队列, 实现了 FIFO 特性, 是生产者/消费者模式的首选, 需要使用轮询 Pull 的方式, 周期性地从队列中取元素。其可以指定元素的容量, 也可以不指定容量, 但是需要注意无界队列使用不当很容易引起 OOM, 建议使用有界队列。其有以下两种使用方式。

- 配合使用 take 和 put。take 是取出元素，如果队列为空则阻塞直到有新的元素放入；put 是放入元素，如果队列已满，则阻塞直到队列有空间。
- 配合使用 poll 和 offer。这里是使用它们带 timeout 的方法。这样，poll 可以在指定的时间内获取元素，超时没有元素则返回；offer 则是在指定时间内放入元素，超时则返回。

## 2) ConcurrentLinkedQueue

非阻塞队列，元素按 FIFO 原则进行排序，采用 CAS 操作，保证元素的一致性。配合使用 offer 和 poll 并使用轮询 Pull 的方式，周期性地从队列中取元素即可实现消息队列的功能。

由于此类是一个无界队列，因此使用的时候需要特别注意消费者的速度要匹配得上生产者的速度，否则会产生 OOM。

## 3) Disruptor

Disruptor 是 LMAX 开源的并发框架，其设计为一个无锁的高性能线程交互的消息处理库，经常作为消息队列来使用。其使用了以下技术来达到低延迟、高性能。

- CAS：避免了使用锁产生的竞争和等待。
- 缓存行填充：避免了伪共享导致的缓存失效问题。
- RingBuffer：一个环形队列，用来在不同线程间传递数据的 Buffer。
- 内存屏障：使用内存屏障（一个 CPU 指令，允许针对数据什么时候对其他进程可见做出假设），避免了锁的开销。

Disruptor 架构如图 6-13 所示。

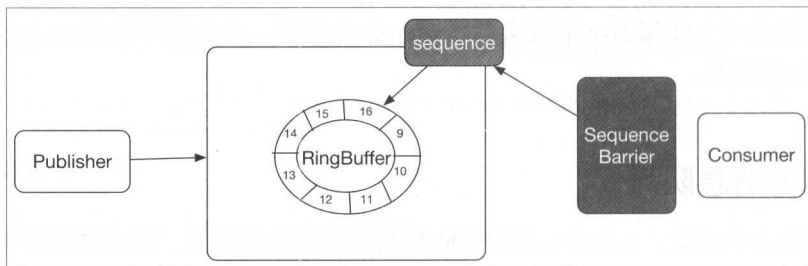


图 6-13

大体流程如下。

- 生产者放入元素时需要申请下一个可写入的空闲槽的 Sequence。

- 生产者拿到空闲 Sequence 后获取对应的槽里的对象并做操作。
- 消费者线程申请下一个可用元素是通过 waitFor Sequence Barrier 来进行的。
- Sequence Barrier 拿到可用 Sequence 后返回给消费者。

消费者使用示例如下：

```
// 事件
class TestEvent {
    private long value;

    public void set(long value) {
        this.value = value;
    }
}

// 事件处理器
class TestEventHandler implements EventHandler<TestEvent> {
    public void onEvent(TestEvent event, long sequence, boolean
    endOfBatch) {
        System.out.println("TestEvent: " + event); // 处理消息
    }
}

// 事件工厂
EventFactory factory = new TestEventFactory(){
    public TestEvent newInstance() {
        return new TestEvent();
    }
};

// 构造 Disruptor
Disruptor<TestEvent> disruptor = new Disruptor<>(factory, 1023,
    Executors.defaultThreadFactory());

// 设置事件处理 Handler
disruptor.handleEventsWith(new TestEventHandler());

// 启动所有相关线程
disruptor.start();
```

生产者如下：

```
// 事件生产者封装，并实现事件的转换
class TestEventProducerWithTranslator {
    private final RingBuffer<TestEvent> ringBuffer;

    public TestEventProducerWithTranslator(RingBuffer<TestEvent>
    ringBuffer) {
        this.ringBuffer = ringBuffer;
    }
}
```

```

private static final EventTranslatorOneArg<TestEvent, ByteBuffer>
TRANSLATOR = new EventTranslatorOneArg<TestEvent, ByteBuffer>() {
    public void translateTo(TestEvent event, long sequence,
ByteBuffer bb) {
        event.set(bb.getLong(0));
    }
};

public void onData(ByteBuffer bb) {
    ringBuffer.publishEvent(TRANSLATOR, bb); // 发布消息
}
}

TestEventProducerWithTranslator producer = new TestEventProducerWithTranslator(disruptor.getRingBuffer());

ByteBuffer bb = ByteBuffer.allocate(8);
bb.putLong(0, 1);
producer.onData(bb); // 发布消息

```

使用 `Disruptor` 需要注意的一点就是，`new Disruptor` 有另外一个构造方法：

```

Disruptor( final EventFactory<T> eventFactory,
           final int ringBufferSize,
           final ThreadFactory threadFactory,
           final ProducerType producerType,
           final WaitStrategy waitStrategy)

```

其中的 `producerType` 指的是生产者模式，包括单生产者和多消费者两种模式。虽然根据单一写原则单生产者模式具有更高的并发性能，但是即使你使用了单生产者模式，`Disruptor` 也不会给你构造单生产者运行环境。

此外，`waitStrategy` 则指的是消费者等待 `ringBuffer` 中的 `Sequence` 可用时使用的等待策略，常用的有以下几种。

- **BlockingWaitStrategy**: 默认的等待策略，使用了锁和竞态条件，是最慢的一种等待策略。适用于关注 CPU 资源大于关注吞吐量和延迟的应用场景。
- **SleepingWaitStrategy**: 先使用一个循环和 `Thread.yield()` 做等待，最后使用 `LockSupport.parkNanos(1L)`，兼顾了性能和 CPU 使用。比较适用于对延迟没有特别关注的场景，如异步日志。
- **YieldingWaitStrategy**: 先做几次循环，再使用 `Thread.yield()` 做等待。适用于低延迟应用且所在机器的 CPU 使用了超线程技术。
- **BusySpinWaitStrategy**: 是一个不停循环等待的策略，也是性能最高的策略。适用于所在机器 CPU 没有使用超线程技术的系统。

# 第 7 章

## Java 编程进阶

根据网络可以找到的资料以及笔者能够打听到的消息，目前国内外著名的几个大型互联网公司的主要语言选型如下。

- Google: C/C++、Go、Python、Java、JavaScript，不得不提的是 Google 贡献给 Java 社区的 Guava 包质量非常高，非常值得学习和使用。
- YouTube、豆瓣: Python。
- Facebook、Yahoo、Flickr、新浪: PHP（优化过的 PHP VM）。
- 网易、阿里、搜狐: Java、PHP、Node.js。
- Twitter: Ruby → Java，之所以如此就在于与 JVM 相比，Ruby 的 runtime 非常慢，并且 Ruby 相对于 Java 来说还是比较小众。不过最近 Twitter 有往 Scala 上迁移的趋势。

可见，虽然最近这些年很多言论都号称 Java 已死或者不久将死，但是 Java 的语言应用占有率一直居高不下。与高性能的 C/C++ 相比，Java 具有 GC 机制，并且没有那让人望而生畏的指针，上手门槛相对较低；而与上手门槛更低的 PHP、Ruby 等脚本语言来说，又比这些脚本语言有性能上的优势（暂且忽略 Facebook 自己开发的 HHVM）。而且，Java 也在不断地吸收其他语言的优势，优化自身的实现和使用。如果说 Java 编程是 Java 工程师最基础的技能点，那么掌握其中的高级特性则是利用 Java 语言优势的关键。这些技能可以提高 Java 工程师的开发效率、代码质量以及 Java 应用的性能。本章就主要讲述以下相关知识。

- **Java 内存管理**：了解 Java 是如何做内存管理的才能从根本上掌握 Java 的编程技巧，避免一些内存问题的出现。
- **Java 网络编程**：了解网络编程模型有助于使用 Java 做网络编程并能够更好地优化实现。

- Java 并发编程：并发是提升应用性能非常关键的手段。
- Java 开发利器：了解 Java 中常用的工具类库，能够大大提升编程开发效率。
- New Java：Java 7、8、9 带来了一些新特性，可以提升开发效率和程序性能。

## 7.1 Java 内存管理

对于 Java 程序员来说，大多数情况下的确无须对内存的分配、释放做太多考虑，对 JVM 也无须有多么深的理解。但是在写程序的过程中却也往往因为这样而造成一些不容易察觉到的内存问题，并且在内存问题出现的时候，也不能很快地定位并解决。因此，了解并掌握 Java 的内存管理是一个合格的 Java 程序员必需的技能，也只有这样才能写出更好的程序，更好地优化程序的性能，避免出现内存问题。

对于 Java 来说，最终要依靠字节码运行在 JVM 上。常见的 JVM 有以下几种。

- Oracle HotSpot。
- Oracle JRockit（原来的 Bean JRockit）。
- IBM J9。
- Dalvik（Android）。
- ART（Android 4.4 后引入）。

其中以 HotSpot 应用最广泛，开源的 OpenJDK 也使用的是这个 JVM。目前 Oracle JDK 的最新 GA 版本已经到了版本 9，但鉴于现在 JDK 8、9 并未完全普及，因此本书仅仅针对 HotSpot 虚拟机的 JDK 7 版本进行讲解。

### 7.1.1 JVM 虚拟机内存

Java 内存管理指的就是对 JVM 虚拟机的内存管理，本节将从内存组成、内存分配过程、对象访问等几个方面来进行介绍。

#### Java 运行时内存区

Java 的运行时内存组成如图 7-1 所示。

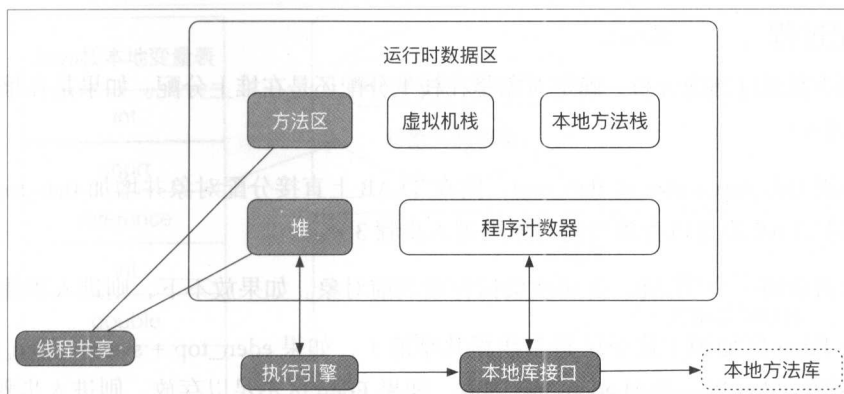


图 7-1

这些组成部分有一些是线程私有的，其他的则是线程共享的。

### 1) 线程私有。

- **程序计数器**：当前线程所执行的字节码的行号指示器。
- **Java 虚拟机栈**：Java 方法执行的内存模型，每个方法被执行时都会创建一个栈帧，存储局部变量表、操作栈、动态链接、方法出口等信息。每个线程都有自己独立的栈空间，线程栈只存储基本类型和对象地址，方法中局部变量存放在线程空间中。
- **本地方法栈**：Native 方法服务。在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

### 2) 线程共享。

- **Java 堆**：存放对象实例，几乎所有的对象实例及其属性都在这里分配内存。此外，JVM 在内存新生代 Eden Space 中开辟了一小块线程私有的区域，称作 TLAB（Thread-Local Allocation Buffer），这也是每个线程的缓冲区，默认设定为占用 Eden Space 的 1%。在编译器做逃逸分析的时候，根据分析结果，决定是在栈上还是在堆上分配内存，如果是在堆上则再分析是否在 TLAB 上分配内存。在 TLAB 上分配由于是线程私有的，因此没有锁的开销，效率比较高。
- **方法区**：存储已经被虚拟机加载的类信息、常量、静态变量、JIT 编译后的代码等数据，也被称作永久代。这里需要注意的是，Java 7 已经把字符串常量池移动到了堆中，在调用 String 的 intern 方法时，如果堆中存在相同的字符串对象，则会直接保存对象的引用，不会重新创建对象。
- **直接内存**：NIO、Native 函数直接分配的堆外内存。DirectBuffer 引用也会使用此部分内存。



## 内存分配过程

1) 编译器通过逃逸分析, 确定对象是在栈上分配还是在堆上分配。如果是在堆上分配, 则跳到步骤 4)。

2) 如果  $\text{tlab\_top} + \text{size} \leq \text{tlab\_end}$ , 则在 TLAB 上直接分配对象并增加  $\text{tlab\_top}$  的值, 如果现有的 TLAB 不足以存放当前对象则进入步骤 3)。

3) 重新申请一个 TLAB, 并再次尝试存放当前对象。如果放不下, 则进入步骤 4)。

4) 在 Eden 区加锁 (这个区是多线程共享的), 如果  $\text{eden\_top} + \text{size} \leq \text{eden\_end}$ , 则将对象存放在 Eden 区, 增加  $\text{eden\_top}$  的值, 如果 Eden 区不足以存放, 则进入步骤 5)。

5) 执行一次 Young GC (Minor GC)。

6) 经过 Young GC 之后, 如果 Eden 区仍然不足以存放当前对象, 则直接分配到老年代。

## 对象访问

Java 是一种面向对象的编程语言, 那么如何通过引用来访问对象呢? 一般有如下两种方式。

- 通过句柄访问 (如图 7-2 所示)。

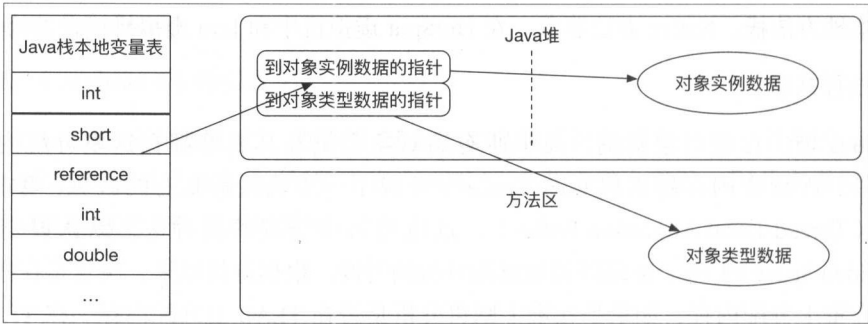


图 7-2

这种方式引用保存的是句柄的地址, 需要先定位句柄, 再定位对象的实例和类型地址。

- 直接指针 (如图 7-3 所示)。

这种方式引用直接保存的是对象实例的地址, 对象实例中保存了类型数据的指针, 这是 HotSpot 虚拟机采用的方式。

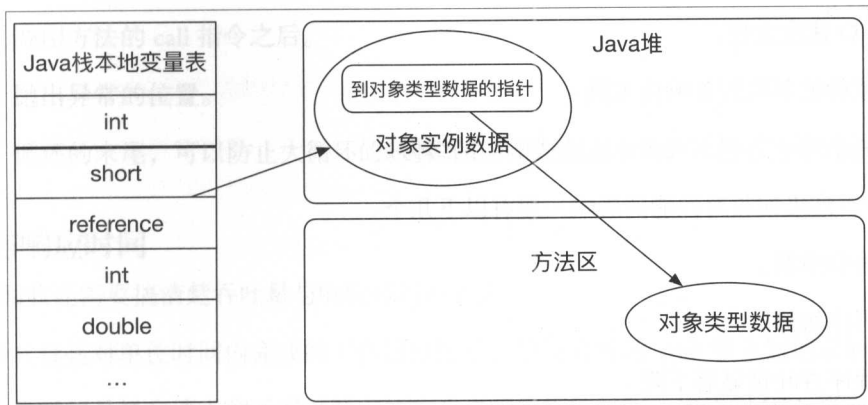


图 7-3

## 内存溢出

在 JVM 申请内存的过程中，会遇到无法申请到足够内存的情况，从而导致内存溢出。一般有以下几种情况。

- 虚拟机栈和本地方法栈溢出。
  - StackOverflowError: 线程请求的栈深度大于虚拟机所允许的最大深度。循环递归会触发这种 OOM。
  - OutOfMemoryError: 虚拟机在扩展栈时无法申请到足够的内存空间，一般可以通过不停地创建线程触发这种 OOM。
- Java 堆溢出: 在创建大量对象并且对象生命周期都很长的情况下，会引发 OutOfMemoryError。
- 方法区溢出: 方法区存放 Class 等元数据信息，如果产生大量的类（使用 CGLIB），那么就会引发此内存溢出，即 OutOfMemoryError:PermGen space，在使用 Hibernate 等动态生成类的框架时会容易引发这种情况。

### 7.1.2 垃圾回收理论

在通常情况下，Java 内存管理就是为了应对网站 / 服务访问慢的问题，慢的原因一般有以下几点。

- 内存: 垃圾回收占用 CPU；放入了太多数据，造成内存泄漏。
- 线程死锁。

- I/O 速度太慢。
- 依赖的其他服务响应太慢。
- 复杂的业务逻辑或者算法造成响应的缓慢。

其中，垃圾回收对性能的影响一般有以下几个。

- 内存泄漏。
- 程序暂停。
- 程序吞吐量显著下降。
- 响应时间变慢。

## 垃圾回收的一些基本概念

- **Concurrent Collector**: 回收的同时可运行其他的工作进程。
- **Parallel Collector**: 使用多 CPU 进行垃圾回收。
- **Stop-the-word(STW)**: 回收时必须暂停其他所有的工作进程。
- **Sticky-reference-count**: 对于使用“引用计数”(reference count)算法的 GC, 如果对象的计数器溢出, 则起不到标记某个对象是垃圾的作用, 这种错误被称为 sticky-reference-count problem, 通常可以增加计数器的位数来减少出现这种问题的概率, 但是那样会占用更多空间。一般如果 GC 算法能迅速清理完对象, 则不容易出现这种问题。
- **Mutator**: mutate 的中文意思是变异, 在 GC 中指一种 JVM 程序, 专门用于更新对象的状态, 也就是让对象“变异”成为另一种类型, 比如变为垃圾。
- **On-the-fly**: On-the-fly 引用计数垃圾回收, 用来描述某个 GC 的类型。此 GC 不用标记而是通过引用计数来识别垃圾。
- **Generational GC**: 这是一种相对于传统的“标记-清理”技术来说, 比较先进的 GC。其特点是把对象分成不同的 Generation, 即分成几代人, 有年轻的, 有年老的。这类 GC 主要利用了计算机程序的一个特点, 即“越年轻的对象越容易死亡”, 也就是存活得越久的对象越有机会存活下去。
- **Safepoint**: 指一些特定的位置, 当线程运行到这些位置时, 线程的一些状态可以被确定, 使 JVM 可以安全地进行一些操作, 比如开始 GC。这些位置主要有如下几种。
  - 一个方法返回前。

- 调用方法的 call 指令之后。
- 抛出异常的位置。
- 循环的末尾，可以防止大循环的时候一直不进入 Safepoint，而其他线程在等待。

## 吞吐量与响应时间

垃圾回收还需要搞清楚吞吐量与响应时间的含义。

- 吞吐量是对单位时间内完成的工作量的度量，如每分钟的 Web 服务器请求数量。
- 响应时间是提交请求和返回该请求的响应之间使用的时间，如访问 Web 页面花费的时间。

吞吐量与访问时间的关系很复杂，有时可能以响应时间为代价而换取较高的吞吐量，而有时又要以吞吐量为代价换取较好的响应时间。而在其他情况下，又有可能同时提高吞吐量和访问时间。

通常，平均响应时间越短，系统吞吐量越大；平均响应时间越长，系统吞吐量越小。但是，系统吞吐量越大，未必平均响应时间越短。因为在某些情况下（例如，不增加任何硬件配置）吞吐量的增大，有时会以牺牲平均响应时间作为代价，以此换取一段时间处理更多的请求。

针对 Java 垃圾回收功能来说，不同的垃圾回收器会不同程度地影响这两个指标。例如，并行的垃圾回收器，其关注的是吞吐量，会在一定程度上牺牲响应时间；而并发的垃圾回收器，则主要关注的是请求的响应时间，会牺牲吞吐量。

如图 7-4 所示，吞吐量优先的垃圾回收器，有可能某一次请求会特别慢；而响应时间优先的垃圾回收器则会尽量使得每一次的响应时间维持在差不多的水平。

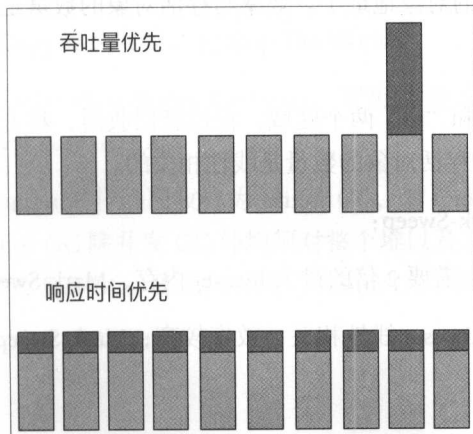


图 7-4

## GC 流程和算法

GC 的一般流程如下。

- 找出堆中活着的对象。
- 释放死对象占用的资源。
- 定期调整活对象的位置。

在找出活着的对象以及释放死对象的情况下可以使用以下算法。

- Mark-Sweep: 标记 - 清除。
- Mark-Sweep-Compact: 标记 - 整理。
- Copying Collector: 复制算法。

### 1) Mark——标记。

从“GC roots”开始扫描（这里的 roots 包括线程栈、静态常量等），给能够沿着 roots 到达的对象标记为“live”，最终所有能够到达的对象都被标记为“live”，而无法到达的对象则被标记为“dead”。效率与存活对象的数量是线性相关的。

### 2) Sweep——清除。

扫描堆，定位到所有的“dead”对象，并将其清理。效率与堆的大小是线性相关的。

### 3) Compact——压缩。

对于对象的清除，会产生一些内存碎片，这时候就需要对这些内存进行压缩、整理。这其中包括：relocate（将存活对象移动到一起，从而释放出连续的可用内存）、remap（回收所有的对象引用指向新的对象地址）。效率与存活对象的数量是线性相关的。

### 4) Copy——复制。

将内存分为“from”和“to”两个区域，在垃圾回收时，将 from 区域的存活对象整体复制到 to 区域中。效率与存活对象的数量是线性相关的。

其中，Copy 对比 Mark-Sweep：

- 内存消耗上：Copy 需要 2 倍的最大 live set 内存，Mark-Sweep 则只需要 1 倍。
- 效率上：Copy 与 live set 线性相关，效率较高；Mark-Sweep 则与堆大小线性相关，效率较低。

分代回收

分代回收是目前比较先进的垃圾回收方案，有以下几个相关理论。

- 分代假设：大部分对象的寿命很短，“朝生夕死”，重点放在对新生代对象的回收，而且新生代通常只占整个空间的一小部分。
- 把新生代里活得很长的对象移动到老年代。
- 只有当老年代满了才去回收。
- 回收效率明显比不分代高。

HotSpot 虚拟机的分代回收，分为一个 Eden 区、两个 Survivor 区以及 Old Generation/Tenured 区，其中 Eden 以及 Survivor 共同组成 New Generation。结构如图 7-5 所示。

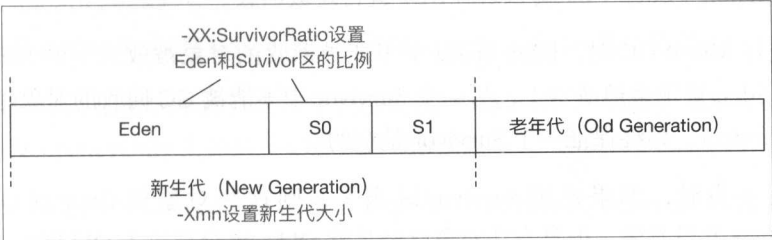


图 7-5

- Eden 区是分配对象的区域。
- Survivor 是 Minor/Younger GC 后存储存活对象的区域。
- Old Generation 区域存储长时间存活的对象，也被称作 Tenured 区。

分代回收中典型的垃圾回收算法组合描述如下。

- 新生代通常使用 Copy 算法回收，会 Stop The World。
- 老年代回收一般采用 Mark-Sweep-Compact，有可能会 Stop The World，也可以是 concurrent 或者部分 concurrent。

通常将对 New Generation 进行的回收称为 Minor GC，对 Old Generation 进行的回收称为 Major GC。但由于 Major GC 除并发 GC 外均须对整个堆以及 Permanent Generation 进行扫描和回收，因此又被称为 Full GC。那么何时进行 Minor GC，何时进行 Major GC？一般的过程如图 7-6 所示。

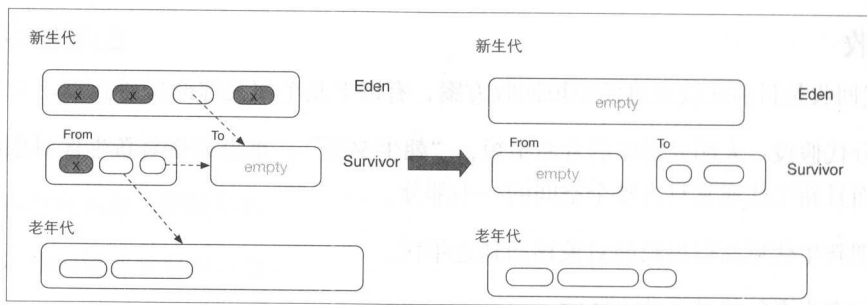


图 7-6

- 对象在 Eden Space 完成内存分配。
- 当 Eden Space 满了，再创建对象，会因为申请不到空间，触发 Minor GC，对 New Generation（Eden + S0 或 Eden + S1）进行垃圾回收。
- 在进行 Minor GC 时，Eden Space 中不能被回收的对象被放入空的 Survivor（S0 或 S1，Eden 肯定会被清空），另一个 Survivor 中不能被 GC 回收的对象也会被放入这个 Survivor，始终保证一个 Survivor 是空的。
- 在上一步时，如果发现 Survivor 区满了，则这些对象被 Copy 到 Old 区，或者 Survivor 并没有满，但是有些对象已经足够 Old，也会被放入 Old 区。
- 当 Old 区被放满之后，进行 Full GC。

但实际操作具体还需要看 JVM 采用的是哪种 GC 方案。新生代 New Generation 的垃圾回收器均是在 Eden Space 分配不下时触发 GC，而老年代 Old Generation 的 GC 有以下几种情况。

- 对于 Serial Old、Parallel Old 而言触发机制为：
  - Old Generation 空间不足。
  - Permanent Generation 空间不足。
  - Minor GC 时的悲观策略。
  - Minor GC 后在 Eden 上分配内存仍然失败。
  - 执行 Heap Dump 时。
  - 外部调用 System.gc。可通过 `-XX:+DisableExplicitGC` 来禁止触发 GC，但需要注意的是，禁用 `System.gc()` 会引起使用 NIO 时的 OOM，所以此选项慎重使用。

- 对于 CMS 而言触发机制为：

- 当 Old Generation 空间使用到一定比率时触发。通过 CMSInitiatingOccupancyFraction 来设置。默认值是根据如下公式计算出来的：

$$((100 - \text{MinHeapFreeRatio}) + (\text{double})(\text{CMSTriggerRatio} * \text{MinHeapFreeRatio}) / 100.0) / 100.0,$$

其中，MinHeapFreeRatio 默认值是 40，CMSTriggerRatio 默认值是 80。

- 当 Permanent Generation 采用 CMS 回收且空间使用到一定比率触发，Permanent Generation 采用 CMS 回收须设置为 -XX:+CMSClassUnloadingEnabled。可通过 CMSInitiatingPermOccupancyFraction 来设置。它是根据如下公式计算出来的：

$$((100 - \text{MinHeapFreeRatio}) + (\text{double})(\text{CMSTriggerPermRatio} * \text{MinHeapFreeRatio}) / 100.0) / 100.0$$

MinHeapFreeRatio 默认值是 40，CMSTriggerPermRatio 默认值是 80。

- HotSpot 根据成本计算决定是否需要执行 CMS GC，可通过 -XX:+UseCmsInitiatingOccupancyOnly 去掉这个动态执行的策略。
- 外部调用 System.gc，且设置了 ExplicitGCInvokesConcurrent 或者 ExplicitGCInvokesConcurrentAndUnloadsClasses。

### 7.1.3 常用垃圾回收器

图 7-7 即为 HotSpot 虚拟机的垃圾回收器组成，后续将分别介绍。

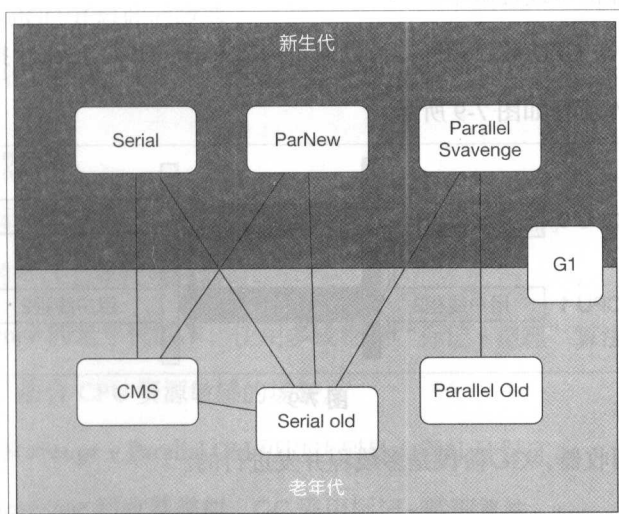


图 7-7



## Serial 回收器

- `-XX:+UserSerialGC` 参数打开此回收器。
- Client 模式下新生代默认的回收器。
- 较长的 Stop The World 时间。
- 简单而高效。如果堆小于 100MB，选择这个回收器即可。

此回收器的工作流程如图 7-8 所示。

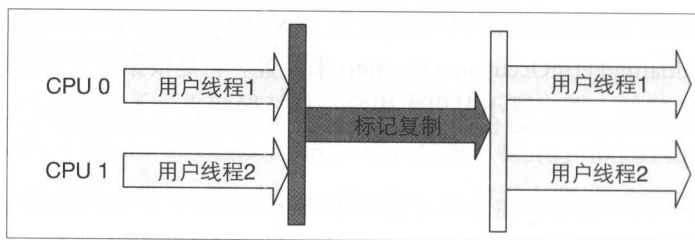


图 7-8

在标记复制阶段会 Stop The World。

## ParNew 回收器

- `-XX:+UserParNewGC` 开启此回收器。
- `+UseConcuMarkSweepGC` 时默认开启。
- Serial 回收器的多线程版本。
- 默认线程数与 CPU 数目相同，可以通过 `-XX:ParallelGCThreads` 指定线程数目。

此回收器的工作流程如图 7-9 所示。

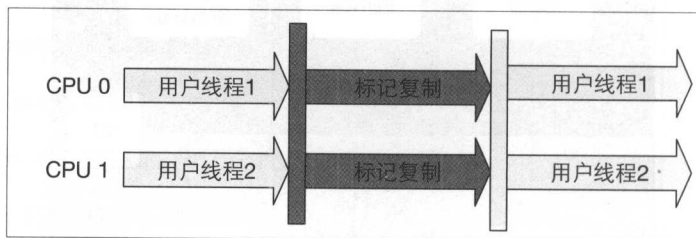


图 7-9

不同于 Serial 回收器，GC 阶段是多线程并发进行的。

## Parallel Scavenge 回收器

- 是 Server 模式的默认新生代回收器。
- 是新生代并行回收器。
- 采用 Copy 算法。
- 主要关注的是吞吐量，吞吐量优先。
- 使用 `-XX:MaxGCPauseMillis` 和 `-XX:GCTimeRatio` 两个参数精确控制吞吐量。
- `-XX:UseAdaptiveSizePolicy` 打开 GC 自适应调节策略，虚拟机会根据当前系统的运行情况回收性能监控信息，动态调整 Survivor 区大小、新生代晋升到老年代的年龄等参数以提供最合适的停顿时间或最大的吞吐量。
- Parallel 是本章介绍的回收器中唯一不进行保留内存（Reserved Memory）释放的回收器，因此当想要达到内存垂直伸缩时，不能使用这个回收器。

其回收流程和 ParNew 类似，不同之处在于其各种参数是自适应调节的。

## Serial Old 回收器

- Serial 的老年代版本。
- Client 模式的默认老年代回收器。
- CMS 回收器的后备预案，Concurrent Mode Failure 时使用。
- `-XX:+UseSerialGC` 开启此回收器。

流程和 Serial 回收器类似，GC 采用标记 - 整理算法。

## Parallel Old 回收器

- `-XX:+UseParallelGC` 和 `-XX:+UseParallelOldGC` 启用此回收器。
- Server 模式的默认老年代回收器。
- Parallel Scavenge 的老年代版本，使用多线程和“标记 - 整理”算法。
- 关注吞吐量，适合 CPU 资源敏感的场景。
- 使用 Parallel Scavenge + Parallel Old 可以达到最大吞吐量保证。

流程和 Parallel Scavenge 回收器类似，GC 采用标记 - 整理算法。

## CMS 回收器

- 并发低停顿回收器，关注的是业务响应时间。
- `-XX:UseConcMarkSweepGC` 开启 CMS 回收器，默认使用 ParNew 作为新生代回收器，SerialOld 作为回收失败的垃圾回收器。
- 以获取最短回收停顿时间为目标的回收器，重视响应速度，希望系统停顿时间最短，适合互联网应用。

其垃圾回收有 4 步，如图 7-10 所示。

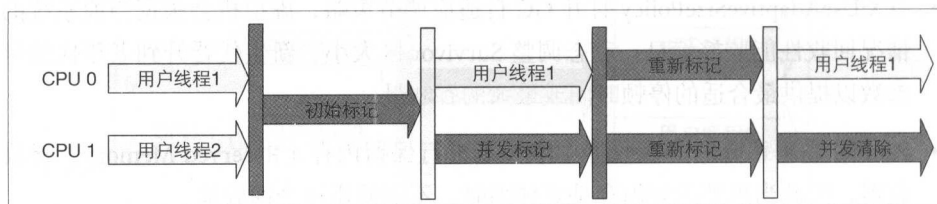


图 7-10

- 初始标记：Stop The World，只标记 GC roots 能直接关联到的对象，速度很快。
- 并发标记：进行 GC roots tracing，与用户线程并发进行。
- 重新标记：Stop The World，修正并发标记期间因程序继续运行导致变动的标记记录。
- 并发清除。

CMS 有以下缺点。

- CMS 是唯一不进行 compact 的垃圾回收器，当 CMS 释放了垃圾对象占用的内存后，它不会把活动对象移动到老年代的一端。
- 对 CPU 资源非常敏感。不会导致线程停顿，但会导致程序变慢，总吞吐量降低。CPU 核数越多越不明显。其适用于业务应用 CPU 使用率不高的场景。
- 无法处理浮动垃圾。可能出现“Concurrent Mode Failure”失败，导致另一次 Full GC，可以通过调整 `-XX:CMSInitiatingOccupancyFraction` 来控制内存占用达到多少时触发 GC。
- 大量空间碎片。这个可以通过设置 `-XX:UseCMSCompacAtFullCollection`（是否在 Full GC 时开启 compact）以及 `-XX:CMSFullGCsBeforeCompaction`（在进行 compact 前 Full GC 的次数）来在一定程度上解决此问题。

## G1 回收器

G1 算法在 Java 6 中还是试验性质的功能，在 Java 7 中已经被正式引入，在 Java 8 中开始有了性能上的巨大提升。这里做一下简单介绍。

- `-XX:+UseG1GC` 可以打开这个垃圾回收器。
- 是并发回收器，关注的是业务响应时间。
- 使用标记 - 清理算法进行 GC。
- 不会产生碎片。
- 可预测的停顿时间。
- 化整为零：将整个 Java 堆划分为多个大小相等的独立区域。
- `-XX:MaxGCPauseMillis=200` 可以设置最大 GC 停顿时间，当然 JVM 并不保证一定能够达到，只是尽力。
- 与 CMS 相比，其适用于大内存的 JVM 垃圾回收（一般情况下以 4GB 堆大小为界）。

## 7.2 Java 网络编程

Java 的网络编程主要指的是网络 I/O 的编程。网络 I/O 会涉及同步、异步、阻塞、非阻塞这几个概念。这几个概念很容易被混淆和误用，因此需要先做几点说明。

- I/O 有内存 I/O、网络 I/O 和磁盘 I/O 共 3 种，通常我们说的 I/O 指的是后两者。
- 阻塞和非阻塞，是函数 / 方法的实现方式，即在数据就绪之前是立刻返回还是等待，即发起 I/O 请求是否会被阻塞。
- 一个网络 I/O 读过程是数据从网卡→内核缓冲区→用户内存的过程。同步与异步的区别主要在于数据从内核缓冲区到用户内存的这个过程需不需要用户进程等待，即实际的 I/O 读写是否阻塞请求进程。

最适合 I/O 模型的例子应该是平常生活中去餐馆吃饭这个场景，下文就结合这个场景来讲解一下经典的几个 I/O 模型。

### 7.2.1 常见网络 I/O 模型

UNIX 环境下的经典 I/O 模型包括同步阻塞、同步非阻塞、I/O 复用、信号驱动以及异步非阻塞共 5 种。

#### 同步阻塞

去餐馆吃饭，点一个自己最爱吃的盖浇饭，然后在原地等着一直到盖浇饭做好，自己端到餐桌就餐。这就是典型的同步阻塞。当厨师给你做饭的时候，你需要一直在那里等着。

在网络编程中，读取客户端的数据需要调用 `recvfrom`。在默认情况下，这个调用会一直阻塞直到数据接收完毕，这就是一个同步阻塞的 I/O 方式。这也是最简单的 I/O 模型，在通常 `fd`（文件描述符）较少、就绪很快的情况下使用是没有问题的。同步阻塞流程如图 7-11 所示。

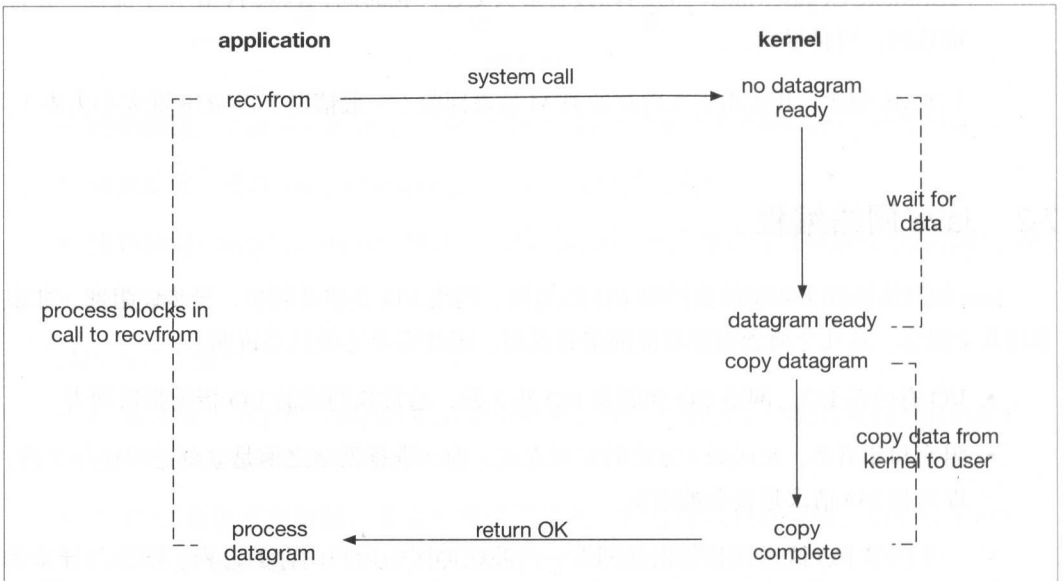


图 7-11

#### 同步非阻塞

接着上面的例子，你每次点完饭就在那里等着，突然有一天你发现自己真傻。于是，你点完之后，就回桌子那里坐着，然后估计差不多了，问老板饭好了没，如果好了就去端，没好的话等一会再去问，依次循环直到饭做好。这就是同步非阻塞。

这种方式在编程时将 `Socket` 设置为 `O_NONBLOCK` 即可。需要提一点：此方式仅针对网络 I/O 有效，对磁盘 I/O 并没有作用。因为本地文件 I/O 就没有被认为是阻塞，我们所

说的网络 I/O 阻塞是因为网络 I/O 有无限阻塞的可能，而本地文件除非被锁住，否则不可能出现无限阻塞的情况，因此只有锁这种情况下，O\_NONBLOCK 才会有作用。而且，在进行磁盘 I/O 时，要么数据就在内核缓冲区中，直接可以返回；要么需要调用物理设备去读取，这时候进程的其他工作都需要等待。因此，后续的 I/O 复用和信号驱动 I/O 对文件 I/O 也是没有意义的。同步非阻塞流程如图 7-12 所示。

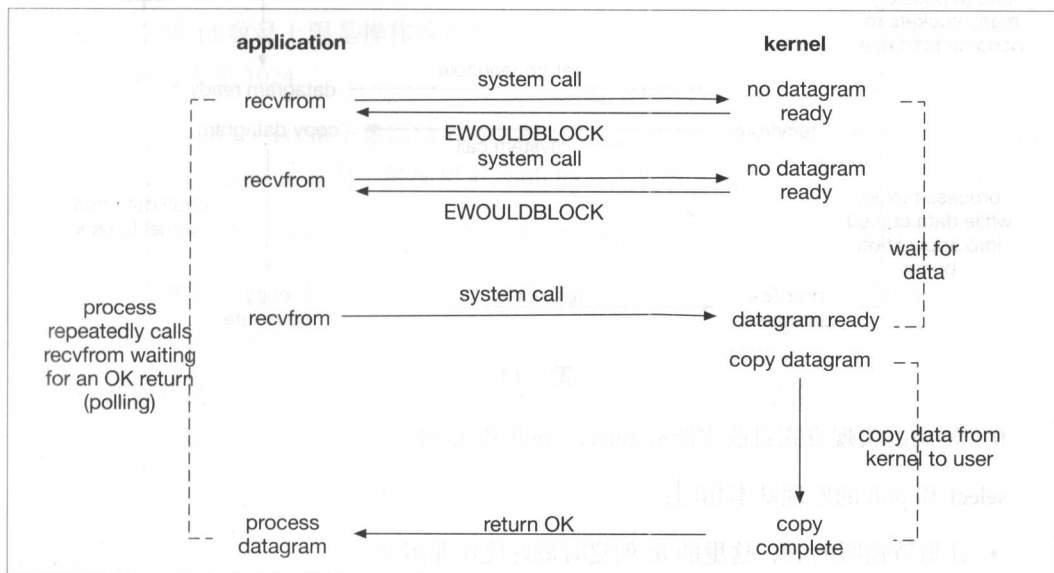


图 7-12

此外，需要说明的一点是，Nginx 和 Node.js 中对于本地文件的 I/O 采用线程的方式模拟非阻塞效果，而对于静态文件的 I/O，使用 Zero-Copy（例如 `sendfile`）的效率是非常高的。

## I/O 复用

接着上面的例子，你点一份饭后循环地问好没好，这显然有点得不偿失，还不如就等在那里直到准备好。但是当你点了好几样饭菜的时候，你每次都去问一下所有饭菜的状态（未做好 / 已做好）肯定比你每次阻塞在那里等着好多了。当然，你问的时候会发阻塞，一直到有准备好的饭菜或者你等得不耐烦（超时）。这就引出了 I/O 复用，也叫多路 I/O 就绪通知。这是一种进程预先告知内核的能力，让内核发现进程指定的一个或多个 I/O 条件就绪了，然后就通知进程，使得一个进程能在一连串的事件上等待。I/O 复用流程如图 7-13 所示。

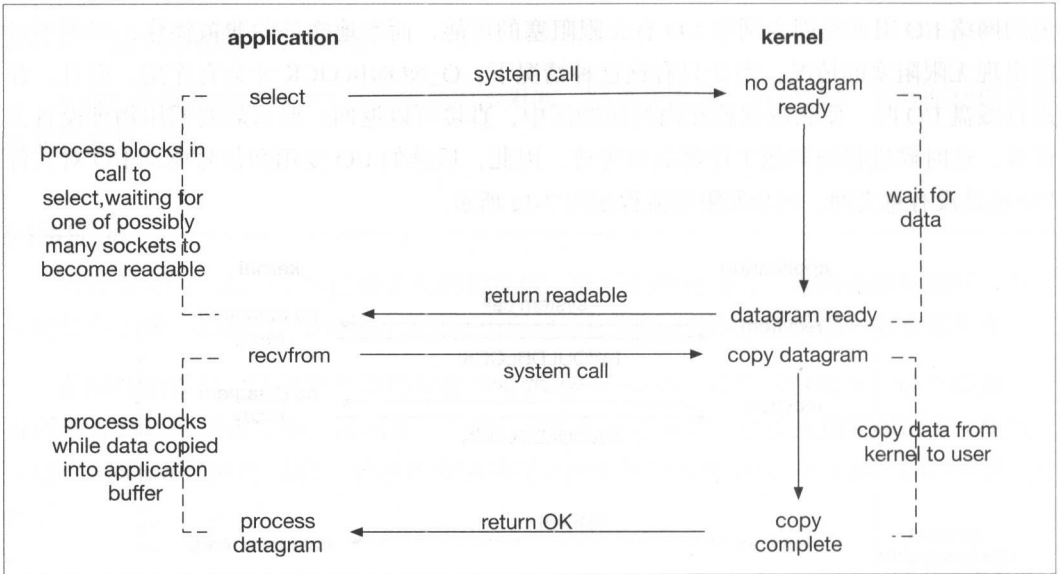


图 7-13

I/O 复用的实现方式目前主要有 select、poll 和 epoll。

select 和 poll 的原理基本相同：

- 注册待侦听的 fd，这里的 fd 创建时最好使用非阻塞。
- 每次调用都去检查这些 fd 的状态，当有一个或者多个 fd 就绪的时候返回。
- 返回结果中包括已就绪和未就绪的 fd。

相比 select，poll 解决了单个进程能够打开的文件描述符数量有限制这个问题：select 受限于 FD\_SIZE，如果要修改则需要修改这个宏重新编译内核；而 poll 通过一个 pollfd 数组向内核传递需要关注的事件，避开了文件描述符数量的限制。

此外，select 和 poll 共同具有的一个很大的缺点就是，包含大量 fd 的数组被整体复制于用户态和内核态地址空间之间，开销会随着 fd 数量增多而线性增大。

select 和 poll 就类似于上面说的就餐方式。但当你每次都去询问时，老板会把所有你点的饭菜都轮询一遍再告诉你情况，当大量饭菜很长时间都不能准备好的情况下这样是很低效的。于是，老板有些不耐烦了，就让厨师每做好一个菜就通知他。这样每次你去问的时候，他会直接把已经准备好的菜告诉你，让你去端。这就是事件驱动 I/O 就绪通知的方式——epoll。

epoll 的出现，解决了 select、poll 的缺点：

- 基于事件驱动的方式，避免了每次都要把所有 fd 都扫描一遍。
- epoll\_wait 只返回就绪的 fd。
- epoll 使用 mmap 内存映射技术避免了内存复制的开销。
- epoll 的 fd 数量上限是操作系统的最大文件句柄数目，这个数目一般和内存有关，通常远大于 1024。

目前，epoll 是 Linux 2.6 下最高效的 I/O 复用方式，也是 Nginx、Node.js 的 I/O 实现方式。而在 FreeBSD 下，kqueue 是另一种类似于 epoll 的 I/O 复用方式。

此外，对于 I/O 复用还有一个水平触发和边缘触发的概念，分别介绍如下。

- **水平触发**：当就绪的 fd 未被用户进程处理时，下一次查询依旧会返回，这是 select 和 poll 的触发方式。
- **边缘触发**：无论就绪的 fd 是否被处理，下一次不再返回。理论上这种方式性能更高，但是实现相当复杂，并且任何意外的丢失事件都会造成请求处理错误。epoll 默认使用水平触发，通过选择相应选项可以使用边缘触发。

## 信号驱动

上面的就餐方式还是需要你每次都去问一下饭菜状况。于是，你不耐烦了，就跟老板说，哪个饭菜好了就通知我一声吧。然后自己坐在桌子那里干自己的事情。更甚者，你可以把手机号留给老板，然后出门，等饭菜好了让老板直接发条短信给你。这类似于信号驱动的 I/O 模型。信号驱动流程如图 7-14 所示。

流程如下。

- 开启套接字信号驱动 I/O 功能。
- 系统调用 sigaction 执行信号处理函数（非阻塞，立刻返回）。
- 数据就绪，生成 sigio 信号，通过信号回调通知应用来读取数据。

这种 I/O 方式存在一个很大的问题：Linux 中信号队列是有限制的，如果超过这个数字就无法读取数据。



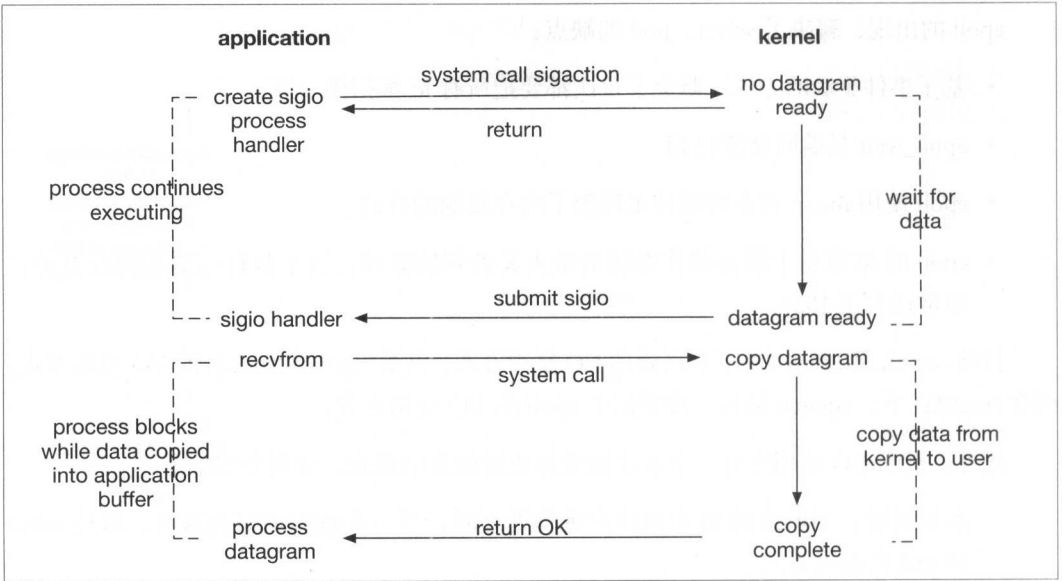


图 7-14

## 异步非阻塞

之前的就餐方式，到最后总是需要你自己把饭菜端到餐桌。这让你也很不耐烦，于是就告诉老板，能不能饭好了直接端到你的面前或者送到你的家里（外卖）。这就是异步非阻塞 I/O，流程如图 7-15 所示。

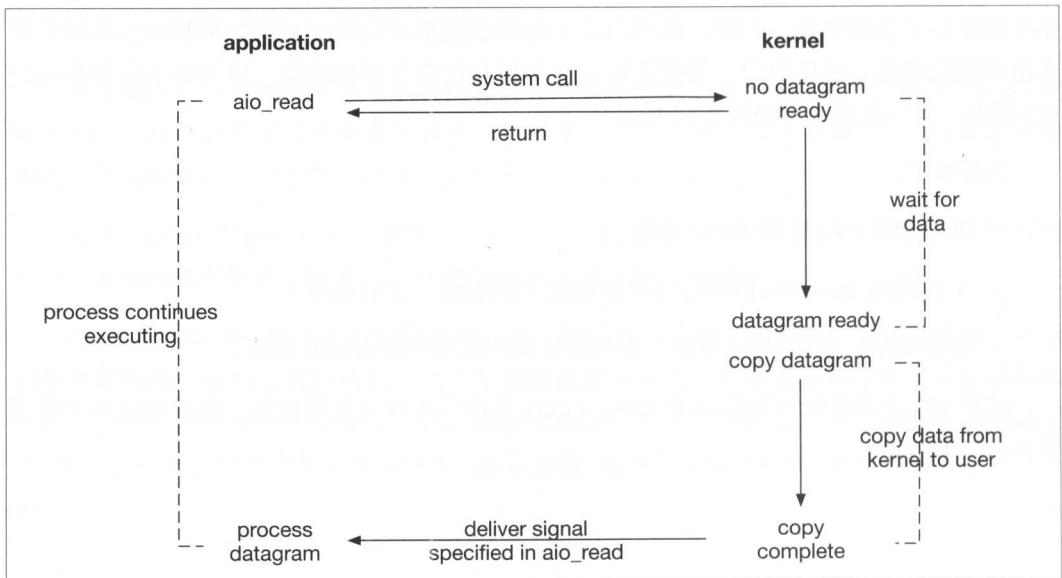


图 7-15

相比信号驱动 I/O、异步 I/O 的主要区别在于：信号驱动由内核告诉我们何时可以开始一个 I/O 操作（数据在内核缓冲区中），而异步 I/O 则由内核通知 I/O 操作何时已经完成（数据已经在用户空间中）。

异步 I/O 又叫作事件驱动 I/O，在 UNIX 中，POSIX1003.1 标准为异步方式访问文件定义了一套库函数，定义了 AIO 的一系列接口。使用 `aio_read` 或者 `aio_write` 发起异步 I/O 操作，使用 `aio_error` 检查正在运行的 I/O 操作的状态。但是其实现没有通过内核而是使用了多线程阻塞。此外，还有 Linux 自己实现的 Native AIO，依赖两个函数 `io_submit` 和 `io_getevents`，虽然 I/O 是非阻塞的，但仍需要主动去获取读 / 写状态。

需要特别注意的是：AIO 是 I/O 处理模式，是一种接口标准，各家操作系统可以实现也可以不实现。目前 Linux 中 AIO 的内核实现只对文件 I/O 有效，如果要实现真正的 AIO，需要用户自己来实现。

### 7.2.2 Java 网络编程模型

前面讲述了 UNIX 环境的 5 种 I/O 模型。基于这 5 种模型，在 Java 中，随着 NIO 和 NIO 2.0（AIO）的引入，一般具有以下几种网络编程模型。

- BIO。
- NIO。
- AIO。

#### BIO

BIO 是一个典型的网络编程模型，是通常我们实现一个服务器端程序的过程，步骤如下。

- 主线程 `accept` 请求阻塞。
- 请求到达，创建新的线程来处理这个套接字，完成对客户端的响应。
- 主线程继续 `accept` 下一个请求。

这种模型有一个很大的问题是：当客户端连接增多时，服务器端创建的线程也会暴涨，系统性能会急剧下降。因此，在此模型的基础上，类似于 Tomcat 的 BIO Connector，采用的是线程池来避免对每一个客户端都创建一个线程。有些地方把这种方式叫作伪异步 I/O（把请求抛到线程池中异步等待处理）。

## NIO

JDK 1.4 开始引入了 NIO 类库，这里的 NIO 指的是 Non-block I/O，主要是使用 Selector 多路复用器来实现的。Selector 在 Linux 等主流操作系统上是通过 epoll 实现的。

NIO 的实现流程，类似于 select：

- 创建 ServerSocketChannel 监听客户端连接并绑定监听端口，设置为非阻塞模式。
- 创建 Reactor 线程，创建多路复用器（Selector）并启动线程。
- 将 ServerSocketChannel 注册到 Reactor 线程的 Selector 上。监听 accept 事件。
- Selector 在线程 run 方法中无限循环轮询准备就绪的 key。
- Selector 监听到新的客户端接入，处理新的请求，完成 TCP 三次握手，建立物理连接。
- 将新的客户端连接注册到 Selector 上，监听读操作。读取客户端发送的网络消息。
- 客户端发送的数据就绪则读取客户端请求，进行处理。

相比 BIO，NIO 的编程非常复杂。

## AIO

JDK 1.7 引入了 NIO 2.0，提供了异步文件通道和异步套接字通道的实现。其底层在 Windows 上是通过 IOCP，在 Linux 上是通过 epoll 来实现的（LinuxAsynchronousChannelProvider.java、UnixAsynchronousServerSocketChannelImpl.java）。

- 创建 AsynchronousServerSocketChannel，绑定监听端口。
- 调用 AsynchronousServerSocketChannel 的 accept 方法，传入自己实现的 CompletionHandler。包括上一步都是非阻塞的。
- 连接传入，回调 CompletionHandler 的 completed 方法，在里面调用 AsynchronousSocketChannel 的 read 方法，传入负责处理数据的 CompletionHandler。
- 数据就绪，触发负责处理数据的 CompletionHandler 的 completed 方法。继续做下一步处理即可。
- 写入操作类似，也需要传入 CompletionHandler。

这种编程模型相比 NIO 有了不少的简化。

## 7.3 Java 并发编程

随着计算机技术的发展, CPU 正从以前通过提升频率转变为通过增加核数来提升整体性能, 而并发就是利用多核特性的关键技术。本节主要讲述 Java 语言下的并发编程。

### 7.3.1 并发原理

要了解并发编程, 需要先了解并发问题产生的原因, 从原理层面理解并发。

#### 并发与并行

并发和并行是看起来差不多而实则大不一样的两个概念。并发指的是同时应对多件事情的能力, 而并行则指同时做多件事情的能力。比如你能够一边吃饭一边看电视这叫作并发, 但是你是不可能同时做多件事情的, 所以不是并行。而一个班级的学生一起打扫教室, 同时扫地、擦黑板以及拖地, 这就是并行。

对于程序员来说通常说的并行大多数情况指的是任务级别的并行, 而除此之外, 计算机在其他层面也有其他的并行实现。

- **位级并行**: 32 位的计算机能够同时处理 32 位数的运算, 而 8 位的计算机却要进行多次计算。
- **指令级并行**: 虽然从表面上看 CPU 都是在串行执行的, 但是内部使用了流水线、乱序执行和猜测执行。
- **数据级并行**: 可以并行地在大量数据上施加同一类操作, 图像处理就是一种非常适合数据级并行的场景。

对于我们常说的任务级并行, 其依赖的基础是计算机的多处理器架构, 主要有以下 3 种。

#### 1) SMP (Symmetric Multi-Processor)

对称多核架构, 也可以叫作统一内存访问架构 (UMA, 是相对于后面的 NUMA 来讲的), 其主要特性就在于所有 CPU 平等地 (无主次或者从属关系) 共享所有资源, 包括内存、I/O、总线等, 如图 7-16 所示。

这种架构下, 所有 CPU 都公用一个总线访问内存, 并且每个 CPU 都有自己的缓存。由于缓存相互独立, 有一个关键的问题即缓存一致性问题, 即如何保证内存中相同地址的数据在各个缓存中保持一致。解决这个问题有很多协议, 常见的 MESI 协议定义了一些缓存读 / 写需要遵循的规范。

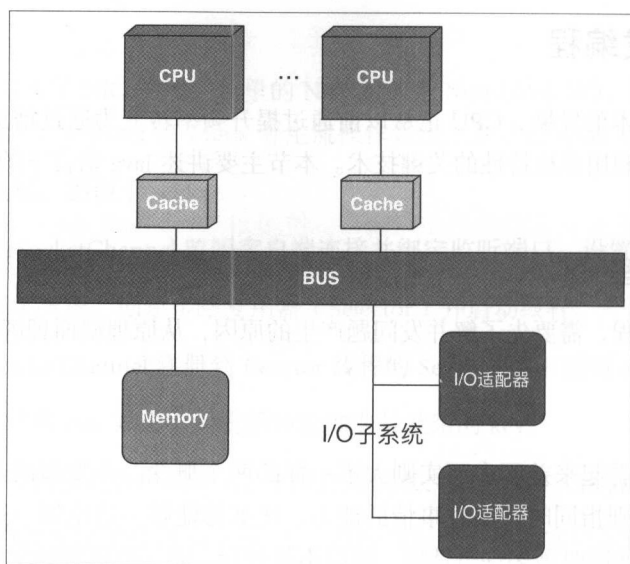


图 7-16

- **Modified:** 本 CPU 写, 则直接写到 Cache, 不产生总线事务; 其他 CPU 写, 则不涉及本 CPU 的 Cache, 其他 CPU 读, 则本 CPU 需要把 Cache line 中的数据提供给它, 而不是让它去读内存。
- **Exclusive:** 只有本 CPU 有该内存的 Cache, 而且和内存一致。本 CPU 的写操作会导致转到 Modified 状态。
- **Shared:** 多个 CPU 都对该内存有 Cache, 而且内容一致。任何一个 CPU 写自己的这个 Cache 都必须通知其他的 CPU。
- **Invalid:** 一旦 Cache line 进入这个状态, CPU 读数据就必须发出总线事务, 从内存读。

通过扩展 CPU 数目可以提升这种架构的性能, 但是相关实验证明, SMP 服务器 CPU 利用率最好的情况是 2~4 个 CPU。

## 2) NUMA (Non-Uniform Memory Access)

非一致性内存访问相对于 SMP 来讲, 其是由多个 CPU 组构成的, 每一个 CPU 组都有自己独立的内存、总线、I/O 等。不同的 CPU 组之间可以通过互连模块互相通信和交互, 使得每一个 CPU 都可以访问整个服务器的所有内存, 但是访问本地内存的效率远远高于远程内存, NUMA 架构如图 7-17 所示。

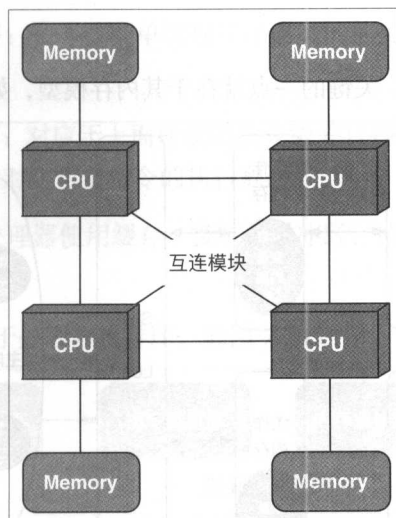


图 7-17

### 3) MPP (Massive Parallel Processing)

大规模并行处理架构也可以叫作分布式内存架构，即 CPU 单元是地理上隔离的，其交互使用网络来进行。每个节点都只能访问自己的本地资源，是一种完全无共享的结构，因而扩展能力最好。从一定意义上来说，这种架构有点类似于现在的 Hadoop 大数据集群（本质上是不同的两个概念）。MPP 架构如图 7-18 所示。

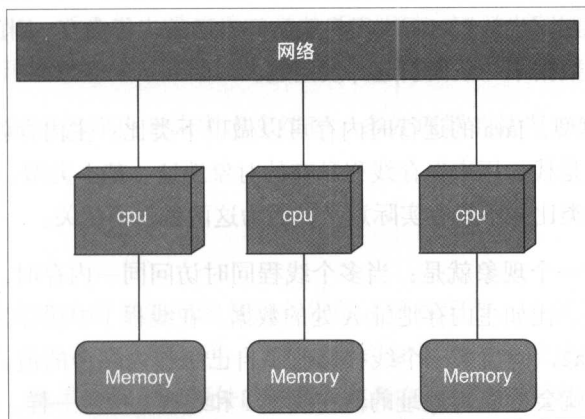


图 7-18

本书涉及的并发是一个宏观上的概念，到了微观层面由于 CPU 数目的限制仍然是串行或者部分串行的。

## Java 内存模型

提到 Java 中的并发问题，关键的一点就在于其内存模型，如图 7-19 所示。

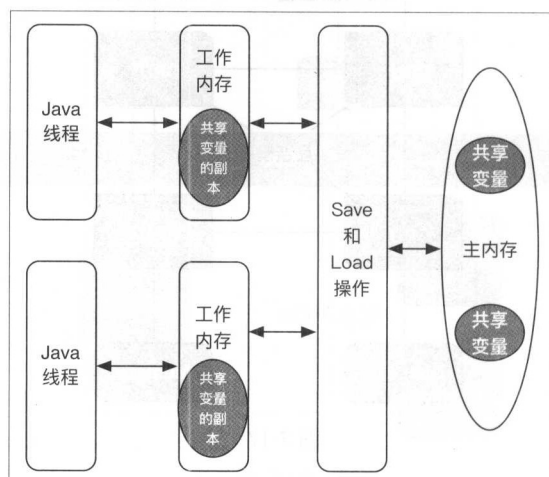


图 7-19

如图 7-19 所示，这有点类似于 SMP，但 Java 内存模型是屏蔽了底层硬件环境的差异而给 Java 程序提供的统一的内存访问模式，可以认为是一种虚拟内存环境。在 Java 程序中，所有线程都共享主内存，但是对于每一个线程都有自己的工作内存（一个虚拟的概念，包括寄存器、栈、写缓冲区、缓存以及其他硬件、编译器优化等），工作内存与主内存通过一些规定的操作来交互同步数据，而线程只能访问自己的工作内存。因此在多线程环境下，很容易造成工作内存数据不一致而引起并发问题。

基于 Java 内存模型，Java 的运行时内存可以做以下类比：主内存就是堆，用来保存对象信息；工作内存就是栈，用来保存线程私有的对象地址、基本类型、局部变量、PC 指针等信息。当然，这个类比并不具有实际意义，因为这两者就不相关。

以上经常造成的一个现象就是：当多个线程同时访问同一内存时，在每个线程的工作内存中的缓存不一致。比如主内存地址 A 处的数据，在线程 1 中缓存为 a1，在线程 2 中缓存为 a2，开始时 a1=a2，但当某一个线程修改了自己工作内存中的值，却没有采取相应的内存同步措施时，就会造成 A 地址的值在线程 1 和线程 2 中不一样。

## 重排序

在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序，可以分为以下 3 种。

- **编译器优化的重排序**：在不改变单线程程序语义的前提下，可以重新安排语句执行顺序。
- **指令级并行的重排序**：对应于上面所说的指令级并行技术。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
- **内存重排序**：由于处理器使用缓存、读 / 写缓冲区，使得加载和存储操作看上去可能是在乱序执行。

指令和内存重排序都属于处理器重排序，如图 7-20 所示。

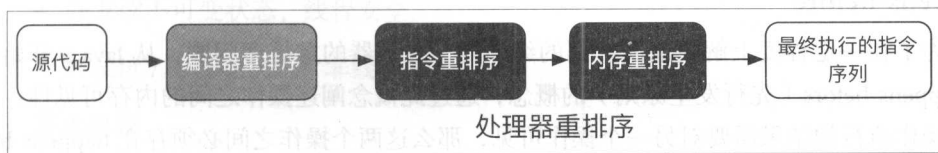


图 7-20

这些基础层面的重排序会遵循以下规范以保证程序的正确性。

- **数据依赖性**：即如果两个操作访问同一个变量，且这两个操作中有一个为写操作，那么这两个操作之间就存在数据依赖性。此时，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。这里需要注意的是，不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。
- **as-if-serial 语义**：不管怎么重排序，单线程程序的执行结果不能被改变。编译器、runtime 和处理器都必须遵守此语义。

编译器和处理器的重排序由于未考虑多线程、多处理器的情况，因此在并发环境下会造成不可预知的问题。

## 并发问题

如前面所述，并发首先带来的问题就是并发的正确性问题，也就是线程安全问题。这里的线程安全指的是“当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那么这个对象是线程安全的。”

此外，多线程之间还存在一个线程同步的问题，即线程之间如何通信协作。

以上两种问题都是 Java 编程中需要解决的。



### 7.3.2 并发思路

除了重排序和 Java 内存模型，可变数据是引起线程不安全的最根本原因。如果一个数据不可变，那么无论多少线程同时访问都是不会产生问题的，这就好比每个线程都拿到了一个副本，但这个副本都是不可变的，也就不可能存在并发问题。针对这些，已经有一些技术方案能够解决线程安全的问题。概括来看，它们都是围绕着在并发过程中如何处理原子性、可见性以及有序性来进行的。

#### happens-before

为了在一定程度上解决前面提到的编译器、处理器的重排序问题，从 Java 5 开始提出了 happens-before（先行发生原则）的概念，通过此概念阐述操作之间的内存可见性：如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须存在 happens-before 关系。这两个操作既可以在一个线程内也可以在不同线程之间。通过此原则，可以判断数据是否存在竞争、线程是否安全。

- **程序次序法则**：在一个线程内如果编码中 A 操作写在 B 操作之前，那么 happens before。
- **监视器锁法则**：对一个监视器的解锁一定发生在后续对同一监视器加锁之前。锁必须是同一锁。
- **volatile 变量法则**：写 volatile 变量一定发生在后续对它的读之前。
- **线程启动法则**：Thread.start 一定发生在线程中的动作之前。
- **线程终结法则**：线程中的任何动作一定发生在括号中的动作之前（其他线程检测到这个线程已经终止，从 Thread.join 调用成功返回，Thread.isAlive() 返回 false）。
- **线程中断法则**：一个线程调用另一个线程的 interrupt 一定发生在另一线程发现中断之前，通过 Thread.interrupted() 方法检测到是否有中断发生。
- **对象终结法则**：一个对象的构造函数结束一定发生在对象的 finalize() 方法之前。
- **传递性**：A 发生在 B 之前，B 发生在 C 之前，A 一定发生在 C 之前。

此原则在一定程度上解决了可见性和有序性的问题，是 Java 内存模型中的“天然的”先行发生关系，无须任何同步器协助就已经存在。如果两个操作不在上述的规则中，也无法通过以上规则推导出来，那么虚拟机就可以随意地对它们进行重排序。需要注意的是，这里的先行发生原则和时间先后顺序之间基本没有关系。

## 原子性

原子性指的是对于对象的操作要么成功要么失败，不会存在中间状态。使用具有原子性对象的方法是线程安全的。

Java 中类型的原子性如下。

- 对象类型

- 对象地址原子读 / 写，线程安全。
- 并发读不可变状态，线程安全。
- 并发读 / 写可变状态，非线程安全。

- 基本类型

- int char 数值读 / 写，线程安全。
- long double 高低位，非线程安全。
- i++ 等组合操作，非线程安全。

以上，对于具有原子性的操作是可以保证线程安全的；对不具有原子性的操作，可以使用 `synchronized` 关键字使其具有原子性。

## 可见性

可见性指的是当一个对象在多个线程工作内存中存有副本时，如果一个内存修改了共享变量，其他线程也能够看到被修改后的值。

- **final**：初始化 `final` 字段确保可见性，这里需要注意 `final` 修饰基本数据类型，可以保证此字段的可见性，但是如果修饰的是对象，那么需要保证此对象是状态不可变的才能保证可见性，即保证对象的每个字段也是 `final` 的。
- **volatile**：此关键字的语义保证了新值能够立即同步到主内存，并且每次使用前都立即从主内存刷新。`volatile` 保证了多线程操作时变量的可见性。
- **synchronized**：在同步块内读 / 写字段也确保了可见性。

如前面所述，`happens-before` 解决了大多数可见性问题。

## 有序性

有序性即保证多个线程对同一对象有序地进行操作。

- happens-before 解决了 Java 中“天然的”有序问题：在一个线程中所有操作都是有序的，其他符合 happens-before 原则的或者可以推导出的操作都是有序的。
- volatile 可以创建内存屏障（指令重排序时不能把后面的指令重排序到内存屏障之前的位置）禁止指令重排序。
- synchronized：“一个变量在同一时刻只允许一条线程对其进行 lock 操作”，使得持有同一锁的两个同步块只能串行地进入。

## 提示

以上，可见使用 volatile 可以在一定程度上解决并发问题，并且由于其开销较小，在其语义能够解决问题的情况下，优先选择 volatile。此外，可以发现 synchronized 貌似是万能的，但是万能的东西通常会伴随着性能问题。

### 7.3.3 并发工具

JDK 自身以及一些第三方类库提供了一些工具类帮助我们解决并发问题，进行并发编程。

## 锁

锁是通过互斥同步来解决线程安全问题的。万能的 synchronized 关键字是锁的一种。除此之外，ReentrantLock 是另一种锁方案。字面上即为可重入锁。这里的可重入指的是：同一线程，外层函数获得锁之后内层递归函数如果仍然有获取该锁的代码，不受影响。synchronized 也是可重入的。

ReentrantLock 与 synchronized 相比，更加灵活，且具有等待可中断、可实现公平锁、可以绑定多个条件等特性。

- 等待可中断：当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情。
- 公平锁：多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁；非公平锁则不保证这一点，在锁被释放时，任何一个等待锁的线程都有机会获得锁。synchronized 中的锁是非公平的，ReentrantLock 默认也是非公平的，但通过配置可以使用公平锁。
- 绑定多个条件：synchronized 中，锁对象的 wait、notify 或者 notifyAll 可以实现一个隐含的条件，如果要和多于一个条件进行关联，那么不得不额外地添加一个锁。而 ReentrantLock 可以通过 newCondition 方法绑定多个条件。

此外, `ReentrantLock` 通过使用 `condition` 的 `await` 和 `signal` 可以做到线程间通信的目的。

## 无锁

除了使用锁解决并发问题, 还有一些无锁技术可以解决并发问题。

- **CAS**: 即 `Compare And Swap`, 这是一种类似于乐观锁的机制。每次更新值的时候都使用旧值与变量的当前值做比较, 如果相同则进行更新, 否则重试直到成功。
- **ThreadLocal**: 本地存储变量, 这样每一个线程都有一份数据的副本, 也就不会存在并发问题了。
- **不可变对象**: 不可变对象自然不会有并发问题。

这里还需要提到 **AQS** (全称 `Abstract Queued Synchronizer`, 基于 **CAS** 来保证线程安全), 是 Java 并发包中的一个类, 提供了一些模板方法供子类实现, 从而实现了不同的同步器 (如 `Sync`、`FairSync`、`NonfairSync` 等)。`ReentrantLock`、`ReentrantReadWriteLock` 以及 `ThreadPoolExecutor` 都使用了 **AQS**。

## 并发集合

并发集合指的就是 Java 自身提供的 `java.util.concurrent` 下的集合类, 以下是常用的几个。

- **ConcurrentHashMap**: 是 `HashMap` 的线程安全版本, 与使用 `Collectons.synchronizedXX` 方法包装不安全的 `HashMap` 相比, `ConcurrentHashMap` 是依据 `bucket` 做的锁, 锁的粒度减小使得性能得到了提高。
- **CopyOnWriteArrayList**: 此类是 `ArrayList` 的线程安全版本, 用了持久化数据结构, 即当写入 `list` 的时候会使用原 `list` 的一个副本来进行操作。
- **LinkedBlockingQueue**: 阻塞队列实现。通过阻塞来解决线程安全问题。

## 同步器

同步器的类也位于 `java.util.concurrent` 包下, 主要有以下几个。

- **CountDownLatch**: 利用它可以实现类似计数器的功能。比如有一个任务 A, 它要等待其他 4 个任务执行完毕之后才能执行, 此时就可以利用 `CountDownLatch` 来实现这种功能。
- **CyclicBarrier**: 字面意思为回环栅栏, 通过它可以实现让一组线程等待至某个状态之后再全部同时执行。叫作回环是因为当所有等待线程都被释放以后, `CyclicBarrier`

可以被复用。我们暂且把这个状态就叫作 barrier，当调用 `await()` 方法之后，线程就处于 barrier。

- Semaphore: 信号量，Semaphore 可以控制同时访问的线程个数，通过 `acquire()` 获取一个许可，如果没有就等待。通过 `release()` 释放一个许可。

## 并发框架

主要的并发框架有以下这几个。

- Executor: 这是 Java 自带的并发框架，封装了并发常用的一些操作。通过调用 Executor 的 `newXXX` 方法可以创建各种 `ExecutorService`（可以看作线程池），包括单线程、固定数目线程池、可扩展线程池、定时调度线程池等，之后使用 `ExecutorService` 的 `execute` 和 `submit` 方法即可运行任务。这里需要注意的一点是，基本所有的 `ExecutorService` 对任务的异常都做了捕获，当你的任务代码抛出异常时，你是拿不到错误的。所以需要你在自己的任务中（`Runnable` 或者 `Callable`）捕获异常并处理。
- Fork/Join: 此框架是 Java 7 带来的并发框架。原理类似于 MapReduce，是一种基于分治法的方案，任务可以递归地分解为子集。这些子集可以并行处理，然后每个子集的结果被归并到一个结果中。和 `ThreadPoolExecutor` 类一样，它也实现了 `Executor` 和 `ExecutorService` 接口。但与我们平常使用 `ThreadPoolExecutor` 相比，其 Work Stealing（工作窃取）可以使得各个线程的负载均衡，不会存在任务分配不均的情况。
- Actor: 一种并发模式，在 JVM 上的实现是 Akka 框架。它的“任其崩溃”哲学在很多场景下都很有意义：每个 Actor 之间都是互相独立、互不影响的，只要考虑最自然的业务逻辑即可，无须做“防御式编程”。除此之外，Actor 也能够做分布式通信工具，作为 RPC 的一种方案。这里需要注意的是，标准的 Actor（如 Erlang 中的）是和线程没有关系的，其是一个独立于操作系统的实体，可以认为是和线程并列的概念。但是 Akka 的实现中，Actor 并不具有线程的特性，无法被系统调度，也无法主动让出 CPU。因此，在 JVM 上，Actor 的使用需要慎重。况且，如图 7-21 所示，Actor 最终的处理依靠的还是线程池。

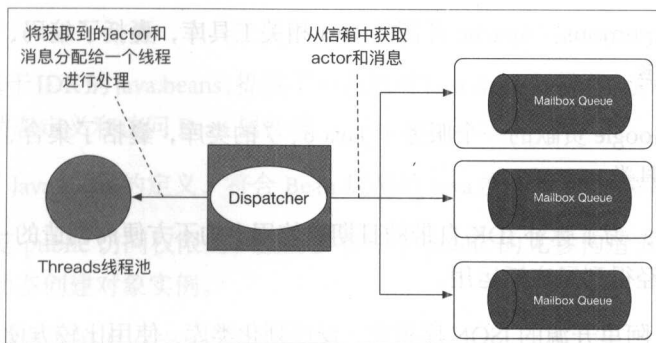


图 7-21

此外，在消息中间件一节讲过的 Disruptor 也是一种高性能并发框架。

### 7.3.4 并发编程建议

并发编程的建议如下。

- 给线程命名，这样可以帮助调试。
- 使用不可变类，所有属性和类都是 `final` 不可变的，可以保证线程安全。
- 总是按照一个全局的固定顺序获取多把锁，可以避免死锁的产生。实例可以参考经典的哲学家就餐问题。
- 最小化同步的范围，而不是将整个方法同步，只对关键部分做同步。
- 分段锁：ConcurrentHashMap 就是采用的这种方式。
- 如果可以，更偏向于使用 `volatile` 而不是 `synchronized`。
- 使用更高层次的并发工具，而不是使用 `wait()` 和 `notify()` 来实现线程间通信，如 `BlockingQueue`、`CountDownLatch` 及 `Semaphore`。
- 优先使用并发集合，而不是对集合进行同步，并发集合能够提供更好的可扩展性。
- 高并发场景下，慎用各种框架，直接到 Servlet 层最好。

## 7.4 Java 开发利器

在 Java 开发中，有很多成熟的开源工具库供大家选用，覆盖了很多常见的但 JDK 中没有提供的功能和使用场景。学习并熟练使用这些类库，能让你在编码过程中少走很多弯路，并且也能学习到很多漂亮的编码方式和风格。比较常用的有以下这几种开发利器。

- Apache Commons: Apache 开源的 Java 相关工具库, 囊括了编码、文本、网络等一系列工具类。
- Guava: Google 贡献的一个服务于 Java 6、7 的类库, 囊括了集合、字符串、缓存等一系列工具类。
- Joda Time: 为了弥补 JDK 自带的日期类使用上的不方便而创造的一个日期 / 时间工具库, 已经得到了广泛运用。
- FastJson: 阿里开源的 JSON 序列化、反序列化类库, 使用比较方便, 性能也比较好。
- Orika: 简单、快速、高效的 Java Bean 映射、复制框架。
- FastUtils: 扩充了 Java 的集合类, 提供了很多快速、压缩、支持基本数据类型的集合类以及大规模集合。
- JCTools: 提供很多并发集合类, 适用于高并发的业务场景。
- Relections: org.relections 提供了一系列对于运行时元数据的查询接口, 大大简化了 Java 自带反射 API 的使用。

此外, 还有其他大厂出品的工具类库如: Twitter 的 Commons、LinkedIn 的 linkedin-utils 等, 基本上也都是对一些常用工具的封装。

本章主要讲述其中最常用的 Apache Commons、Google Guava、Joda Time、FastJson 以及 Orika。

### 7.4.1 Apache 工具库——Apache Commons

Apache Commons 是 Apache 开源的一个可重用 Java 组件库, 其包含了多达 50 个子项目。其中常用的有以下几个组件。

- BeanUtils: 可以对 Java Bean 进行各种操作, 复制对象、属性。
- Codec: 处理常用的编码 / 解码方法的工具类包等。
- Collections: 扩展 Java 集合框架的操作。
- I/O: 输入 / 输出工具的封装。
- Lang: Java 基本对象 (java.lang) 方法的工具类包。
- HttpClient: 低层次对 HTTP 协议操作的封装, 提供 HTTP 客户端与服务器端的各种通信操作。

## BeanUtils

BeanUtils 基于 JDK 的 `java.beans`，提供了一系列对 Java Bean 的操作：读取（`get`）和设置（`set`）Bean 属性值、动态定义和访问 Bean 属性等。

先介绍一下 Java Bean 的定义，符合 Bean 规范的 Java 类需要符合以下要求。

- 类必须是 `public` 访问权限的，且需要有一个 `public` 的无参构造方法，方便利用 Java 的反射动态创建对象实例。
- Bean 的属性都是私有字段。
- Bean 的属性值只能通过 `setter` 方法设置。
- 读取 Bean 的属性需要通过 `getter` 方法。

对于 Bean 的操作，主要通过 BeanUtils 这个类。

BeanUtils 将 property 分成 `simple`（简单类型：String、Int）、`indexed`（索引类型：数组、ArrayList）以及 `Mapped`（Map 类型）共 3 种类型，可以直接 `get` 和 `set` Java Bean 中的一个属性的值。

需要注意的是，该类其实最终调用的是 `BeanUtilsBean`，但由于 `BeanUtilsBean 2` 继承了 `BeanUtilsBean` 并进行了升级，提供了任何类型到字符串的转换操作。因此，建议直接使用 `BeanUtilsBean 2`。

```
BeanUtilsBean beanUtilsBean = new BeanUtilsBean2();
beanUtilsBean.getConvertUtils().register(false, false, 0);
// 错误不抛出异常、不使用 NULL 做默认值，数组的默认大小为 0
```

```
User user = new User();
beanUtilsBean.setProperty(user, "name", "testName");// 设置属性的值
beanUtilsBean.getProperty(user, "name");// 获取属性的值
```

需要注意的是，如果类型是 `indexed`，那么属性名 `[index]` 可以直接获取某个元素的值，而对于 `Mapped` 类型，属性名（key 值）可以获取某一个 key 对应的 value。

此外，还提供了复制 Bean 的功能：

```
User user = new User();
user.setName("test");
User user2 = new User();

beanUtilsBean.copyProperties(user2, user);
User user3= (User)beanUtilsBean.cloneBean(user);
```

但这里的复制是浅复制，两个 Bean 的同一个属性可能拥有同一个对象的引用。



还有 Map 和 Bean 之间的转换:

```
Map<String, Object> map = beanUtilsBean.describe(user); // bean->map
beanUtilsBean.populate(user, map) // map->bean
```

此外, 还有一个 PropertyUtils, 其和 BeanUtils 功能几乎一致。不同的是 BeanUtils 在对 Bean 赋值时会进行自动类型转化, 只要属性名相同, 类型会尝试转换, 而 PropertyUtils 则会报错。

## Codec

常用的解码、编码方法封装, 包括 Base64、MD5、Sha1、URL。

### 1) Base64

```
Base64.encodeBase64String(byte[] binaryData);
Base64.decodeBase64(String base64String);
```

### 2) MD5

```
DigestUtils.md5Hex(final byte[] data);
```

### 3) Sha1

```
DigestUtils.sha1Hex(final byte[] data);
```

### 4) URL

```
URLCodec.encode(final String str);
URLCodec.decode(final String str);
```

## Collections

Collections 为 JDK 的集合类提供了更丰富的工具类、接口以及实现。最新是 4 版本, 包名修改为: org.apache.commons.collections4。

1) CollectionUtils: 提供了一些方面的操作方法, 如判断集合非空, 对集合的并集、差集、交集的操作。如下:

```
List<String> list = getList();
List<String> list2 = getList2();

if(CollectionUtils.isNotEmpty(list)){ // 判断非空
    CollectionUtils.union(list, list2) // 并集
    CollectionUtils.subtract(list, list2) // 差集
    CollectionUtils.retainAll(list, list2) // 交集
}
```

2) 提供了一些新的集合类型。如下:

```
// 得到集合里按顺序存放的 key 之后的某一 key
OrderedMap map = new LinkedHashMap();
map.put("1", "1");
map.put("2", "1");
map.put("3", "1");
map.firstKey(); // returns "1"
map.nextKey("1"); // returns "2"
```

```
// 双向 Map
Bidimap bidi = new TreeBidimap();
bidi.put("6", "6");
bidi.get("6"); // returns "6"
bidi.getKey("6"); // returns "6"
```

## I/O

提供了一些 I/O 工具类, 是对 java.io 的扩展, 可以非常方便地操作文件。

1) IOUtils: 对 I/O Stream 操作的封装。如下:

```
InputStream is = new URL( "http://baidu.cim" ).openStream();
try{
    IOUtils.toString(is, "utf-8");
    IOUtils.readLines(is, "utf-8");
}finally {
    IOUtils.closeQuietly(is);
}
```

2) FileUtils: 对文件操作的封装。如下:

```
File file = new File("/data/data.txt");
List lines = FileUtils.readLines(file, "UTF-8"); // 读取成字符串集合
byte[] fileBytes = FileUtils.readFileToByteArray(file); // 读取成字节数组
FileUtils.writeByteArrayToFile(file, fileBytes); // 字节写入文件
FileUtils.writeStringToFile(file, "test"); // 字符串写入文件
```

3) FileSystemUtils: 对文件系统的操作封装。如下:

```
FileSystemUtils.freeSpaceKb("/data"); // 查看相应路径的剩余空间
```

## Lang

一些公共的工具集合, 涵盖了字符串操作、字符操作、JVM 交互操作、归类、异常和位域校验等。现在最新为 3 版本, 包名改成了 org.apache.commons.lang3。

### 1) StringUtils 和 StringEscapeUtils

StringUtils 继承自 Object，是 NULL safe 的，即遇到 NULL 的 String 对象会把它处理掉而不抛出异常；StringEscapeUtils 是对字符串做转义的工具类，包括 HTML、JS、XML 等。如下：

```
String str = ...;

StringUtils.isEmpty(str); // 判断字符串为空，多个连续空格不为空
StringUtils.isBlank(str); // 判断字符串为空，多个连续空格为空

StringUtils.trim(str);
// 以 strip 开头的方法都是 trim 方法的扩展，不过可以自定义 stripChars，不局限于空白符。

StringUtils.equals(str, "test"); // 支持 NULL

StringUtils.contains("str", "test"); // 子字符串匹配

StringUtils.split(str, ";"); // 根据字符、字符串分隔字符串
StringUtils.join(new String[]{"1", "2"}, "-"); // 根据字符连接字符串

StringEscapeUtils.escapeHtml4(str); // 对字符串中的 HTML 标签做转义
```

需要注意的是，其 split 方法，相比字符串自带的 split 方法使用正则表达式，此方法直接使用了完整的字符串来做匹配，且会丢弃空字符串。

### 2) ArrayUtils

ArrayUtils 是一个对数组进行特殊处理的类。ArrayUtils 扩展了 JDK 中的 Arrays，提供了更多的功能。如下：

```
String[] strs = new String[]{"1", "4", "2"};

ArrayUtils.nullToEmpty(strs); // 如果数组为 NULL，则返回长度为 0 的数组
ArrayUtils.reverse(strs); // 反转数组

ArrayUtils.addAll(strs, "3"); // 数组添加元素
```

需要注意的是，使用 addAll 添加元素，是需要数组复制的，慎用。

### 3) RandomUtils 和 RandomStringUtils

提供了生成随机数、字符串的操作封装。如下：

```
RandomUtils.nextInt(0, 10); // 随机生成 1 个整数，从 0 到 10，不包括 10。
RandomStringUtils.random(3); // 随机生成 3 个字母的字符串。
```

需要注意的是，这两个类都使用了 Random 类，但却是伪随机的，在要求严格的环境下，尽量不要使用这两个类，而使用 SecureRandom。

#### 4) NumberUtils

为数字提供了一些操作封装，是 NULL safe 的。如下：

```
String numberStr = "123";
long n = NumberUtils.toLong(numberStr);
// 将字符串转化为 long，如果字符串格式不对或者为 NULL，则返回 0，并不会抛出异常
long max = NumberUtils.max(new Long[]{1L, 5L, 10L}); // 计算数组最大值
```

#### 5) DateUtils 和 DateFormatUtils

是对日期、时间操作的封装。

DateUtils 提供了很多日期计算：

```
DateUtils.addDays(new Date(), 3); // 计算 3 天后的时间
DateUtils.addHours(new Date(), 3); // 3 个小时后的时间
```

```
DateUtils.truncate(new Date(), Calendar.HOUR); // 截断日期到小时，后面的分、秒都为 0
```

DateFormatUtils 提供了 Date 到字符串表示的操作：

```
DateFormatUtils.format(new Date(), "yyyyMMdd"); // 以 yyyyMMdd 的格式输出日期
```

#### 6) MethodUtils

通过此工具类可以调用类的方法，实现原理基于反射：

```
MethodUtils.invokeStaticMethod(StringUtils.class, "isNotBlank", "test");
// 调用静态方法
MethodUtils.invokeMethod(user, "getName"); // 调用实例方法
```

#### 7) Stopwatch

StopWatch 是一个秒表类：

```
StopWatch stopWatch = new Stopwatch();
stopWatch.start(); // 开始计时
stopWatch.split(); // 截断每一次的分段计时
stopWatch.getSplitTime(); // 获取分段计时
stopWatch.suspend(); // 暂停秒表
stopWatch.resume(); // 恢复计时
stopWatch.stop(); // 停止秒表
stopWatch.getTime(); // 获得总共计时
```

#### 8) ImmutablePair 和 ImmutableTriple

这两个类都是不可变的，经常用在返回值是 2 个或者 3 个的场景下，是对多返回值的通用封装。如下：

```
ImmutablePair pair = ImmutablePair.of(user, user1);
pair.getLeft();
```

```

pair.getRight();
ImmutableTriple triple = ImmutableTriple.of(user,user1,user2);
triple.getLeft();
triple.getMiddle();
triple.getRight();

```

## HttpClient

提供 HTTP 客户端与服务器端的各种通信操作，包括支持各种 HTTP method、SSL 连接、Cookie、Session 保持等。此工具类现在已经从 Apache Commons 移到 Apache HttpComponents 中，并且包名被改为 org.apache.http。

### 1) 连接池

HttpClient 提供了 HTTP 连接池的支持，连接池依赖 HTTP 1.1 的 Keep Alive 机制，对 HTTP 1.0 需要做兼容配置。此外，也支持 HTTPS 请求。需要注意的是，使用连接池能够减少频繁创建、销毁连接的消耗并提高性能，但是由于连接池是有锁的，为了提升并发性能，最好对每一个服务都创建一个连接池。

```

RegistryBuilder<ConnectionSocketFactory> schemeRegistry = RegistryBuilder.
create();
schemeRegistry.register("http", PlainConnectionSocketFactory.
getSocketFactory());

```

// 对 HTTPS 的支持

```

SSLContext sslcontext = SSLContext.getInstance("TLS");
sslcontext.init(new KeyManager[0], new TrustManager[]{new
SimpleTrustManager()}, null);
SSLConnectionSocketFactory sf = new SSLConnectionSocketFactory(sslcontext);
schemeRegistry.register("https", sf);

```

// 连接池配置

```

PoolingHttpClientConnectionManager pool = new PoolingHttpClientConnection
Manager(schemeRegistry.build());
pool.setMaxTotal(1000); // 最大连接数支持
pool.setDefaultMaxPerRoute(100); // 每一个路由的最大连接数
pool.setDefaultSocketConfig(SocketConfig.custom().setSoTimeout(5000).
build());

```

### 2) HttpRequestBase

HttpClient 支持 HTTP 的各种方法，都是 HttpRequestBase 的子类。

- HttpGet。
- HttpPost。
- HttpPut。

- `HttpDelete`。

以上类都有一个参数为 URL 字符串的构造方法。此外,对 Post、Put 这种需要传递数据的方法, `HttpClient` 是使用 `HttpEntity` 来实现的。常用的几个 `HttpEntity` 如下。

- `UrlEncodedFormEntity`: 最常见的表单提交, `Content-type` 为 `application/x-www-form-urlencoded`。
- `MultipartFormEntity`: 提交文件时常用的方式, `Content-type` 为 `multipart/form-data`。
- `StringEntity`: 自包含的 `Entity`, 传递 JSON 数据时可以使用。

```
StringEntity entity = new StringEntity("{\"name\":\"test\"}", "UTF-8");
entity.setContentType("application/json")
```

```
HttpPost method = new HttpPost(url);
method.setEntity(entity);
```

### 3) Cookie

Cookie 的支持需要依赖 `CookieStore`。 `HttpClientUtil` 内置了 `BasicCookieStore`。如下:

```
CookieStore cookieStore = new BasicCookieStore()
cookieStore.addCookie(Cookie cookie);
List<Cookie> cookies = cookieStore.getCookies();
```

### 4) HttpClientBuilder

`HttpClient` 的创建依赖于 `HttpClientBuilder`, 能够对 `HttpClient` 的 Cookie 以及 Connect Timeout、Socket Timeout 及 Keep Alive 的策略进行配置。如下:

```
HttpClientBuilder httpClientBuilder = HttpClients.custom().setDefaultCookieStore(cookieStore); // Cookie 支持
httpClientBuilder.setConnectionManager(pool); // 设置连接池
httpClientBuilder.setDefaultSocketConfig(pool.getDefaultSocketConfig());
httpClientBuilder.setDefaultRequestConfig(
    RequestConfig.custom()
        .setConnectTimeout(3000)
        .setSocketTimeout(5000)
        .build());

// 对 Keep Alive 策略配置
httpClientBuilder.setKeepAliveStrategy(new ConnectionKeepAliveStrategy() {
    public long getKeepAliveDuration(HttpResponse response,
    HttpContext context) {
        HeaderElementIterator it = new BasicHeaderElementIterator(
        response.headerIterator(HTTP.CONN_KEEP_ALIVE));
        while (it.hasNext()) {
```

```

        HeaderElement he = it.nextElement();
        String param = he.getName();
        String value = he.getValue();
        if (value != null && param.equalsIgnoreCase("timeout")) {
            try {
                return Long.parseLong(value) * 1000;
            } catch (NumberFormatException ignore) {
            }
        }
        // 否则保持活动 5 秒
        return 5 * 1000;
    }
});

```

```
HttpClient httpClient = httpClientBuilder.build();
```

### 5) 使用

```

method.setProtocolVersion(HttpVersion.HTTP_1_1); // 设置使用 HTTP 1.1
request.addHeader("User-Agent", agentHeader); // 设置 ua
request.addHeader("Connection", "keep-alive");//为了 Keep Alive 支持 HTTP 1.0

```

```
HttpResponse res = httpClient.execute(request);
```

```
byte[] byteResult = EntityUtils.toByteArray(res.getEntity());
```

最终可以通过返回的 `HttpResponse` 拿到接口返回的 `body`、`header` 等信息。

此外, Apache `HttpComponents` 还提供了 `AsyncHttpClient` 用于异步通信场景, 使用流程和 `HttpClient` 类似, 不同之处如下。

1) 连接池管理器多了 I/O 线程的配置且连接的 `Registry` 不同。如下:

```

Registry<SchemeIOSessionStrategy> sessionStrategyRegistry =
RegistryBuilder
    .<SchemeIOSessionStrategy>create()
    .register("http", NoopIOSessionStrategy.INSTANCE)
    .register("https", new SSLIOSessionStrategy(SSLContexts.
createDefault()))
    .build();

// 配置 I/O 线程
IOReactorConfig ioReactorConfig = IOReactorConfig.custom()
    .setIoThreadCount(Runtime.getRuntime().availableProcessors())
    .build();

// 设置连接池
ConnectingIOReactor ioReactor;
ioReactor = new DefaultConnectingIOReactor(ioReactorConfig);
PoolingNHttpClientConnectionManager conMgr = new

```

```
PoolingNHttpClientConnectionManager(ioReactor, null,
sessionStrategyRegistry, null);
conMgr.setMaxTotal(100);
conMgr.setDefaultMaxPerRoute(100);
```

// 连接配置: 忽略传输错误, 默认编码使用 UTF8

```
ConnectionConfig connectionConfig = ConnectionConfig.custom()
    .setMalformedInputAction(CodingErrorAction.IGNORE)
    .setUnmappableInputAction(CodingErrorAction.IGNORE)
    .setCharset(Consts.UTF_8).build();
conMgr.setDefaultConnectionConfig(connectionConfig);
```

## 2) 使用 CloseableHttpAsyncClient。如下:

```
RequestConfig requestConfig = RequestConfig.custom()
    .setConnectTimeout(3000)
    .setSocketTimeout(1000).build();
```

```
CloseableHttpAsyncClient asyncClient = HttpAsyncClients.custom()
    .setConnectionManager(conMgr)
    .setDefaultCookieStore(new BasicCookieStore())
    .setDefaultRequestConfig(requestConfig)
    .build();
```

## 3) 使用时, 需要先启动 Client, 并且提供了回调使用方式。如下:

```
asyncClient.start(); // 需要先启动 Client
```

// 通过 future 获取结果

```
HttpGet httpGet = new HttpGet("http://www.baidu.com");
Future<HttpResponse> responseFuture = asyncClient.execute(httpGet, null);
try {
    HttpResponse httpResponse = responseFuture.get();
    HttpEntity httpEntity = httpResponse.getEntity();
    ...
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

// 回调方式获取结果

```
final HttpGet httpGet2 = new HttpGet("http://www.baidu.com");
asyncClient.execute(httpGet2, new FutureCallback<HttpResponse>() {
    @Override
    public void completed(HttpResponse httpResponse) {
        HttpEntity httpEntity = httpResponse.getEntity();
        ...
    }

    @Override
    public void failed(Exception e) {
    }
}
```



```

@Override
public void cancelled() {

}

});

...

asyncClient.close();

```

需要注意的是，无论是 `HttpClient` 还是 `AsyncHttpClient`，连接池都是有锁的。虽然支持对每一个 `route` 设置最大连接数，但如果是高并发场景，最好对于每一个服务都创建一个单独的 `HttpClient` 实例，使用不同的连接池。

## 7.4.2 Google 工具库——Guava

Guava 是 Google 开源的一个涵盖了字符串处理、缓存、并发库、事件总线、I/O 等常用操作的 Java 核心库，也是 Google 自己很多 Java 项目依赖的工具库。其中的 Guava Cache 缓存已经在 5.3.1 节介绍过。

### 1) Preconditions

对条件做前置判断，经常用在方法的最前面，用来对参数进行校验，不符合则抛出异常：

```

Preconditions.checkNotNull(user != null, "user null error"); //user 为 NULL
则抛出 IllegalArgumentException
Preconditions.checkNotNull(user); //user 为 NULL 则抛出 NullPointerException

```

### 2) Optional

使用 `Optional<T>` 表示可能为 `NULL` 的 `T` 类型引用，能够显著地降低代码抛出 `NullPointerException` 异常的可能。其中有 `of` 方法和 `fromNullable`，前者传入的值不能为空，后者则可以传入 `NULL` 值，表示引用缺失。提供了 `isPresent()` 方法判断是否引用缺失，在调用 `get` 之前务必先调用 `isPresent()`。但 `Optional` 不能乱用，建议仅仅用在对外的 API 和接口的返回值上。如下：

```

String str = ...;
Optional<String> optional = Optional.fromNullable(str);
if(optional.isPresent()){
    String tmp = optional.get();
}

str = optional.or("default string");
str = optional.or(new Supplier<String>() {
@Override

```

```

    public String get() {
        return "default string";
    }
});
str = optional.orNull();

// 对 optional 中的 value 做转换操作
optional.transform(new Function<String, Object>() {
    @Override
    public Object apply(String input) {
        return "transformed string";
    }
});

```

### 3) Objects 和 MoreObjects

是对 Java 中 Object 的操作的扩展，后者是对前者的升级，都是 NULL safe 的，其包括非空判断、相等判断、空值处理、hashCode 计算等。如下：

```

User user = new User();
User user1 = new User();
...

Objects.equal(user, user1); //equal 判断
Objects.hashCode(user); // 获取对象实例的哈希值
MoreObjects.toStringHelper(user).add("name", "testName").toString();
// 辅助编写类的 toString 方法
MoreObjects.firstNonNull(user, new User());
// 取第一个非空的实例，可以用于设置第一个元素为 NULL 时的默认值。

```

### 4) ComparisonChain

提供了链式的比较器，执行比较操作直至发现非零结果，之后的比较将被忽略：

```

ComparisonChain.start()
    .compare(user.getName(), user1.getName())
    .compare(user.getAge(), user1.getAge())
    .result();

```

### 5) Strings、Joiner、Splitter、CaseFormat

这几个类都是对字符串的操作，包括分割、连接、格式转换等：

```

Strings.nullToEmpty(str); // 如果字符串为 NULL，则转换为空字符串
Strings.repeat(str, 3); // 重复字符串成新的字符串

Joiner joiner = Joiner.on(";").skipNulls();
joiner.join("1", null, "2", "3", "4"); // 使用 ; 拼接字符串

Splitter.on(';')
    .trimResults()
    .omitEmptyStrings()

```

```
.split("1;2;3;4"); // 分隔字符串
```

```
CaseFormat.LOWER_UNDERSCORE.to(CaseFormat.LOWER_CAMEL, "user_name");
// 将字符串从 low underscore 命名格式转换为 low camel 命名格式。
```

其中的命名格式转换还支持 LOWER\_HYPHEN、UPPER\_CAMEL 以及 UPPER\_UNDERSCORE。

#### 6) ImmutableList、Multiset、Multimap

这些是 Guava 实现的新的集合类型。

ImmutableList 是不可变集合（保证线程绝对安全）的一种。除此之外，还有 ImmutableSet、ImmutableMap，用法都类似：

```
ImmutableList<String> list = ImmutableList.of("1","2","3");
```

Multiset 可以多次添加相等的元素，主要统计给定元素的个数：

```
Multiset<String> set = HashMultiset.create();
set.add("1");
set.add("1");
System.out.println(set.count("1"));
```

Multimap 是一个 key 映射多值的 Map，类似于 Map<K, List<V>>、Map<K, Set<V>>。主要使用其两个子类：ListMultimap 和 SetMultimap，前者允许重复值，后者则不允许。

```
Multimap<String,String> multimap = ArrayListMultimap.create();
multimap.put("test","1");
multimap.put("test","2");
multimap.get("test"); //[ "1", "2" ];
```

#### 7) Lists 和 Maps

JDK 默认提供了 Collections 作为集合工具类。Guava 针对其没有提供的集合工具操作做了扩展和实现。Lists、Maps 是 list 和 Map 的工具类。最大的特性是提供的静态工厂方法相比先创建 ArrayList 再添加元素或者设置参数的初始化方式要简单、优雅很多。如下：

```
Lists.newArrayList("1", "2", "3");
Lists.newArrayListWithCapacity(100);
```

#### 8) ListenableFuture

ListenableFuture 继承了 JDK concurrent 包下的 Future 接口，可以大大简化并发逻辑的编写，可以注册回调方法，在运算（多线程执行）完成的时候进行调用，或者在运算（多线程执行）完成后立即执行。如下：

```

ListeningExecutorService service = MoreExecutors.
    listeningDecorator(Executors.newFixedThreadPool(10));
ListenableFuture future = service.submit(new Callable<String>() {
    @Override
    public String call() throws Exception {
        return "result";
    }
});
Futures.addCallback(future, new FutureCallback() {
    @Override
    public void onSuccess(Object result) {
        ...
    }

    @Override
    public void onFailure(Throwable t) {
        ...
    }
});

```

还支持链式操作:

```

Futures.transform(future, new AsyncFunction() {
    @Override
    public ListenableFuture apply(Object input) throws Exception {
        return ...;
    }
}, new Executor() {
    @Override
    public void execute(Runnable command) {
    }
});

```

## 9) EventBus

Guava 实现的事件总线机制, 为了替代通用型的发布 / 订阅模型, 不适用于进程间通信。

EventBus 定义了事件生产者和事件监听者的角色, 类似于生产者和消费者。

- **事件生产者:** 管理和追踪监听者以及向监听者分发事件。
- **事件监听者:** 注册到总线上, 按照 Event 监听。只要以 ChangeEvent 为唯一参数创建方法, 并用 Subscribe 注解标记, 即可实现一个监听者。注册到 EventBus 上即可开始监听 Event。

```

class EventBusListener {
    @Subscribe
    public void recordCustomerChange(ChangeEvent e) {
        System.out.println(e.getSource());
    }
}

```

```
EventBus eventBus = new EventBus();
eventBus.register(new EventBusListener());

eventBus.post(new ChangeEvent("test event"));
```

此外, Guava 还提供了散列、函数式风格、I/O、区间、数学运算、Service 框架、BloomFilter 等许多非常有用的工具类, 都大大提高了开发效率和代码质量。其中, Guava 的 Optional、Objects 等由于用得非常广泛, Java 8 也做了类似的实现。

### 7.4.3 最好用的时间库——Joda Time

JDK 自带的 Date、Calendar 类使用起来非常麻烦, 并且日期与字符串之间的转换很慢且非线程安全。Joda Time 就是为了解决这些问题而创造的日期/时间库, 使用起来非常简单、方便。和之前 Apache Commons 提供的 DateUtils 相比, 如果想继续使用 Java 日期, 可以选择 DateUtils; 如果想彻底改变, 可以使用 Joda Time。

#### 1) 初始化时间。

```
DateTime dateTime=new DateTime(2017, 6, 21, 18, 00,0); //2017.06.21
18:00:00
```

#### 2) 输出格式化字符串。

```
dateTime.toString("yyyy-MM-dd");
```

#### 3) 解析时间字符串。

```
DateTimeFormatter format = DateTimeFormat .forPattern("yyyy-MM-dd");
DateTime dateTime = DateTime.parse("2017-06-21", format);
```

#### 4) 时间计算。

```
dateTime.plusDays(1) // 增加天
    .plusYears(1) // 增加年
    .plusMonths(1) // 增加月
    .plusWeeks(1) // 增加星期
    .minusMillis(1) // 减分钟
    .minusHours(1) // 减小时
    .minusSeconds(1); // 减秒数
```

```
DateTime.Property month = dateTime.monthOfYear();
month.isLeap(); // 判断是否是闰月
```

#### 5) 与 Java Date 转换。

```
dateTime = new DateTime(new Date());
dateTime = new DateTime(Calendar.getInstance());
Date date = dateTime.toDate();
```

## 7.4.4 高效 JSON 处理库——FastJson

FastJson 是阿里巴巴开源的 JSON 处理器，官方测试称性能超过 MappingJackson，使用起来比较简单方便。

### 1) 序列化。

```
User user = new User();
user.setName("testUser");
user.setGender("M");
String userJson = JSON.toJSONString(user);
```

### 2) 反序列化。

```
user = JSON.parseObject(str, User.class);
JSONObject jo = JSON.parseObject("{\"name\":\"test\"}");
```

### 3) 构造 JSONObject 以及取值。

```
JSONObject jo = new JSONObject();
jo.put("name", "test");
jo.getString("name");
jo.getString("nickName"); // 返回为 NULL，不会抛出异常
```

### 4) 属性名称转化。

很多时候会遇到 JSON 字符串中的属性和 Java Bean 中的属性名称不一致的情况。这时候如果直接调用，则会出错，需要做名称转换，可以使用 @JSONField 注解配置别名：

```
@JSONField(name = "nick")
private String nickName;
```

此外，FastJson 默认提供了 JSON 属性 Low Underscore 到 Java Bean 字段 Low Camel 命名的转化。如果想要在实现序列化的时候进行 Low Camel 的转换，除了可以使用 @JSONField，还可以使用 SerializeConfig，设置其 PropertyNamingStrategy 即可。同样地，ParseConfig 也能设置此策略。如下：

```
SerializeConfig config = new SerializeConfig();
config.propertyNamingStrategy = PropertyNamingStrategy.SnakeCase;
String str = JSON.toJSONString(user, config);
System.out.println(str); // {"nick_name":"testNick"}
```

PropertyNamingStrategy 还支持 KebabCase（连接单词）、PascalCase（大写字母开头）以及 CamelCase（驼峰）。

### 5) JSONPath。

FastJson 1.2.0 之后提供了 JSONPath，取值非常方便，类似于 XPath。其主要是为了简化取值逻辑，方便嵌套取值、过滤取值、获取集合长度等。如下：

```
String jsonStr = "{\"name\":\"testName\",\"interests\":[\"music\",\"basketball\"],\" +
    \"notes\":[{\"title\":\"note1\",\"contentLength\":200},{\"title\":\"note2\",\"contentLength\":100}]}";
JSONObject jsonObject1 = JSON.parseObject(jsonStr);
System.out.println(JSONPath.eval(jsonObject1, "$.interests.size()"));
// 集合长度
System.out.println(JSONPath.eval(jsonObject1, "$.interests[0]"));
// 集合取值
System.out.println(JSONPath.eval(jsonObject1, "$.notes[contentLength > 100].title")); // 集合过滤取值
System.out.println(JSONPath.eval(jsonObject1, "$.notes['title']")); // 只取某一个属性的值
```

使用 FastJson 时需要注意 FastJson 在序列化和反序列化时默认是开启 ASM 的（Android 系统下不会开启）。可以通过下面的代码关闭：

```
SerializeConfig.getGlobalInstance().setAsmEnable(false);
// 序列化的时候关闭 ASM
ParserConfig.getGlobalInstance().setAsmEnable(false);
// 反序列化的时候关闭 ASM
```

## 7.4.5 高效 Bean 映射框架——Orika

Orika 是一个快速、高效的 Java Bean 映射框架，主要用于在 VO、PO 等各种 Bean 之间复制属性，并且是深复制。相比 7.4.1 节中提到的 BeanUtils 使用反射，Orika 是使用代码生成进行复制的。因此其性能好于 BeanUtils 和 Dozer（使用反射，对反射信息做了缓存）。如下：

```
MapperFactory mapperFactory = new DefaultMapperFactory.Builder().build();
MapperFacade mapper = mapperFactory.getMapperFacade();
```

```
User user = new User();
user.setName("test");
User user1 = mapper.map(user, User.class);
```

不同类之间的复制，如果属性名不一致，可以通过自定义映射来复制，属性名相同则可以直接复制：

```
MapperFactory mapperFactory = new DefaultMapperFactory.Builder().build();
mapperFactory.classMap(User.class, TestUser.class)
    .field("name", "testName")
    .byDefault()
    .register();
```

```
MapperFacade mapper = mapperFactory.getMapperFacade();
```

```
User user = new User();
user.setName("test");
TestUser testUser = mapper.map(user, TestUser.class);
```

## 7.5 Java 新版本的特性

这里所说的 New Java 主要指的是 Java 6 以后的版本，包括 Java 7、Java 8 以及 Java 9。

### 7.5.1 Java 7

不像 JDK 5 和 8，JDK 7 只是一个小版本，没有做太大的改动，其只引入了一些小的改变和特性。其中 Fork/Join 框架和内存的一些改动在前面的章节中已经讲过。其他的改动主要如下。

#### 1) 改进的通用实例创建类型推断。

```
Map<String, List<String>> map = new HashMap<>();
```

如上钻石运算符 <> 的引入使得后面的创建实例不必指定类型，可以从引用的声明中推断类型。

#### 2) 自动资源管理。

Java 中的 InputStream、SQL Connection 等资源都是需要手动关闭的，以前我们必须通过 try final 来实现。Java 7 提供了 try-with-resource 这个新的语言特性，允许 try 语句本身申请更多的资源，这些资源作用于 try 代码块并自动关闭。只要资源实现了 AutoClosable 接口即可。如下：

```
try (BufferedReader br = new BufferedReader(new FileReader("/data/data.txt"))) {
    br.read();
    ...
}
```

#### 3) switch 语句支持字符串。

```
String type = "text";
switch (type){
    case "text":
        System.out.println("text type");
        break;
    case "image":
        System.out.println("image type");
        break;
}
```

需要注意的是，最终 switch 还是编译成对字符串 hashCode 的 switch，然后再做字符串比较。



4) 允许在同一个 catch 块中捕获多个异常。

```
try {
    ...
} catch (FileNotFoundException | AccessException e) {
    e.printStackTrace();
}
```

5) Path 和 Files。

Java 7 中文件 I/O 发生了很大的变化, 专门引入了很多新的类来简化对文件的操作。其中最核心的特性包括 Path、Paths 以及 Files。这些类的出现是为了取代原来的基于 java.io.File 的文件 I/O 操作方式。

Path 用来取代 File 来表示文件路径和文件, 而 Paths 是其辅助工具类:

```
Path path = Paths.get("/data/test.dat");
Path path1 = FileSystems.getDefault().getPath("/data", "test.dat");
```

Files 利用 Path 做文件创建、读取、写入等各种操作, 几乎包含了所有可能的文件操作, 使用起来非常方便。

- 创建、读 / 写文件。

```
if(!Files.exists(path)){
    Files.createFile(path);
}
```

```
BufferedReader reader = Files.newBufferedReader(path1,
StandardCharsets.UTF_8);
// 获取文件的 BufferedReader 来读取文件内容
```

```
BufferedWriter writer = Files.newBufferedWriter(Paths.get("/data/
test.txt"), StandardCharsets.UTF_8); // 获取文件的 BufferedWriter 来写
文件
```

- 遍历文件夹。

```
Path dir = Paths.get("/data");
DirectoryStream<Path> stream = Files.newDirectoryStream(dir)
for(Path e : stream){
    System.out.println(e.getFileName());
}

Stream<Path> stream = Files.list(dir);
Iterator<Path> it = stream.iterator();
while(it.hasNext()){
    Path curPath = it.next();
    System.out.println(curPath.getFileName());
}
```

- 复制文件。

```
Files.copy(Path source, Path target, CopyOption options);
Files.copy(InputStream in, Path target, CopyOption options);
Files.copy(Path source, OutputStream out);
```

## 7.5.2 Java 8

Java 8 是一个创新版本，带来了很多崭新的特性，改动也比较大。

### 1) 接口的默认方法。

Java 8 允许开发者通过使用关键字 `default` 向接口中加入非抽象方法。这一新特性被称为扩展方法。如下：

```
public interface TestUserService {
    default void test() {
        ...
    }
}
```

此外，也允许向接口添加静态方法：

```
public interface TestUserService {
    static String testStatic() {
        ...
    }
}
```

### 2) Optional。

Java 借鉴 Guava 的 `Optional` 类实现了自己的 `Optional`，除了方法名做了一些改动，用法基本一致。和 Guava 中的 `Optional.transform` 一样，Java 8 的 `Optional` 也提供了 `map` 和 `flatMap`，可以对 `Optional` 对象做一系列转换操作，并且还提供了 `filter` 方法来做过滤。如下：

```
User user = new User();
user.setName("testUser");

Optional<User> optional = Optional.of(user);
user = optional.orElse(new User());
user = optional.orElseThrow(RuntimeException::new);
user = optional.orElseGet(User::new);

Optional<TestUser> testUserOptional =
    optional.filter(u -> u.getName() != null)
        .map(u -> {
            TestUser testUser = new TestUser();
            testUser.setName(u.getName());
```

```

        return testUser;
    });

Optional<User> userOptional = testUserOptional.flatMap(tu -> {
    User curUser = new User();
    curUser.setName(tu.getName());

    return Optional.of(curUser);
});

```

map 和 flatMap 的区别在于前者传入的 mapper Function 返回值可以是任何值而后者则必须是 Optional。

### 3) Effective final。

Java 8 引入了一个叫作 Effective final 的特性，即隐形推导某个变量是否是 final。比如，在使用 Runnable 匿名类新建线程的时候，如果 run 方法里使用了外边的变量，那么之前的版本必须声明为 final，而在 Java 8 中，不需要声明，但必须确保此变量的确是初始化后没有再改变过的。如下：

```

List<String> list = ..;

new Thread(new Runnable() {
    @Override
    public void run() {
        list.add("test");
    }
});

```

### 4) Lambda 表达式。

Lambda 表达式是 Java 8 最主要的特性，其使得 Java 可以使用函数式编程，大大简化了代码行数。如下：

```

List<Long> list = Lists.newArrayList();
list.sort((n1, n2) -> Long.compare(n2, n1));

```

其中 sort 部分的参数就是一个 Lambda 表达式，相比之前的匿名内部类，代码变得简短和便于阅读了。一个 Lambda 表达式由参数列表、箭头和函数体组成。函数体可以是一个表达式，也可以是一个代码块。

这里需要提到一个概念，其叫作函数接口（Functional Interface），指的是仅仅包含一个抽象方法的接口，可以认为任何一个 Lambda 表达式都可以等价转换为对应的函数式接口，可以将任意只包含一个抽象方法的接口用作 Lambda 表示式，但是使用 @FunctionalInterface 有助于编译器检查函数接口的合法性。Runnable 就是一个函数接口。如下：

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

```
Runnable run = () -> System.out.println(); // 打印一个换行符
```

除了 Runnable 外，Java 8 自带的几个常用函数接口如下。

- Predicate: 接收一个参数并返回 Boolean。
- Consumer: 接收一个参数，不返回值。
- Function: 接收一个参数并产生一个结果。
- Supplier: 不接收参数，对于给定的泛型类型产生一个实例。

使用方法引用的话，上面的代码能够进一步简化：

```
Runnable run = System.out::println;
```

这里的方法引用是用关键字 :: 来传递方法和构造函数的引用，形式为类名 :: 方法名，主要分为如下 4 种。

- 静态方法引用: Integer::valueOf。
- 实例方法引用: System.out::println。
- 构造方法引用: User::new。
- 某个类型的任意对象的实例方法引用: User::getName。

## 5) Stream。

Stream 也是 Java 8 引入的一个非常重要的特性，其代表着一系列可以在其上进行多种操作的元素。这些操作可以是连续的，也可以是中断的，中断操作返回操作结果，而连续操作返回流本身，这样就可以形成链式风格的流式操作。

流是创建在数据源上的，java.util.Collection、list 集合和 set 集合都是数据源的一种，产生的流可以是串行的，也可以是并行的。通过集合类的 Stream 方法可以产生串行流，parallelStream 方法可以产生并行流。

一个 Stream 操作如下：

```
List<String> stringList = Lists.newArrayList("1", "0", "5", null);
List<Integer> intList = stringList.stream() // 创建 Stream
    .filter(Objects::nonNull) // 过滤 NULL
    .map(Integer::valueOf) // 转换 Stream
    .filter(e -> e > 0) // 过滤
```

```
// .reduce((e1, e2) -> e1 + e2) // 消减  
.collect(Collectors.toList()); // 聚合
```

Stream 支持的主要连续操作如下。

- filter: 接受一个 predicate 来过滤流中的所有元素。
- sorted: 返回流的已排序版本。
- map: 通过指定的函数将流中的每一个元素转变为另外的对象。
- flatMap: 每个元素转换得到的是 Stream 对象, 会把子 Stream 中的元素压缩到父集合。
- peek: 生成一个包含原 Stream 的所有元素的新 Stream。
- limit: 对一个 Stream 进行截断操作。
- skip: 返回一个丢弃原 Stream 的前 N 个元素后剩下元素组成的新 Stream。

Stream 支持的主要中断操作如下。

- reduce: 使用指定的函数对流中元素实施消减, 返回一个包括所有被消减元素的 Optional。
- collect: 把流中的元素聚合到其他数据结构中。
- match: anyMatch、allMatch 等各种匹配操作可以用来检测是否某种 predicate 和流中元素相匹配。
- count: 返回流中的元素数量。
- findFirst: 返回 Stream 中的第一个元素。
- max 和 min: 使用给定的比较器 (Operator), 返回 Stream 中的最大值 / 最小值。

使用 Stream 有如下几点需要注意。

- 流并不存储值, 它只是某数据源的一个视图, 对流的操作会产生一个结果, 但流的数据源不会被修改。
- 使用 Stream 时, 要遵循先做 filter 再 map 的原则。
- 多数 Stream 操作 (包括过滤 filter、映射 map、排序 sorted 以及去重 distinct) 都以惰性方式实现。
- 不同于其他 Stream 操作, flatMap 不是惰性计算的。
- Stream 只能被“消费”一次, 一旦遍历过就会失效。

- 慎重使用并行 Stream，其底层使用的是 ForkJoinPool 的 commonPool。在不做任何配置的情况下，所有并行流都共用同一个线程池（默认线程数为机器 CPU 数目，可通过系统属性 `java.util.concurrent.ForkJoinPool.common.parallelism` 设置），而且此线程池还会被其他机制依赖。

## 6) Map。

Java 8 对 Map 也做了一些改动，包括方法的增加以及底层实现的改动：

```
Map<String, String> map = new HashMap<>();
map.putIfAbsent("1", "a");
map.forEach((k, v) -> {
    System.out.println(k + v);
});
map.computeIfAbsent("2", e -> e + "1");
map.computeIfPresent("1", (k, v) -> k + v);

map.remove("1", "a")

map.getDefault("2", "b");

map.merge("1", "a", (k, v) -> k + v);
```

- `putIfAbsent`：当 key 没有对应的 value 时才放入新的值，可以防止旧值被覆盖。
- `forEach`：方便了对 map 做遍历。
- `computeIfAbsent`：当 key 没有对应的 value 时才计算生成新的值。
- `computeIfPresent`：当 key 有对应的 value 时，使用 key 和 value 生成新的 value。
- `remove(k,v)`：仅当 k 对应的 value 等于 v 时，才会删除 k。
- `getOrDefault`：获取不到值时使用传入的默认值。
- `merge`：如果 key 对应的值存在，那么将对应的 value 和传入的 value 使用后面的函数合并后作为新值，否则设置为传入的 value。

此外，Java 8 对 HashMap 的实现做了优化，主要的一点是在解决冲突时候，当链表大于 8 时会使用红黑树存储，这样极大地加快了冲突时的查询速度。

使用 Map 还有一点需要注意，ConcurrentHashMap 的 `keySet` 方法被修改了，如果用到这个方法且声明的时候使用的是 ConcurrentHashMap，那么使用 JDK 8 时会报错，可以使用 `ConcurrentMap` 声明或者不要使用此方法。

## 7) Date API。

鉴于原有 Java 时间 API 的各种问题, Java 8 在 java.time 包下提供了新的 date 和 time 的 API, 是由 Joda Time 的作者写的。

使用 LocalDateTime 表示日期、时间, 和时区无关, 可以灵活地创建、进行时间计算、输出 / 解析格式字符串。

```
LocalDateTime.now() // 当前事件
LocalDateTime dateTime = LocalDateTime.of(2017, Month.JUNE, 21, 12, 0, 0);
//2017-6-21 12:00:00

dateTime.plus(1, ChronoUnit.DAYS); // 后面的 1 天
dateTime.plusHours(1); // 后面的 1 小时
dateTime.minus(1, ChronoUnit.HOURS); // 前面的 1 小时
dateTime.truncatedTo(ChronoUnit.HOURS); // 截断时间到小时

dateTime.format(DateTimeFormatter.ofPattern("yyyy-MM-dd")); // 输出格式化字符串
dateTime = LocalDateTime.parse("2016-06-21 12:00:00", DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")); // 解析格式化字符串
```

还有两个类 LocalTime 和 LocalDate, 前者表示时间, 后者表示日期, 都是不可变的, 工作原理类似于 LocalDateTime。

和之前的 java.text.DateFormat 相比, DateTimeFormatter 都是不可变且线程安全的, 可以放心地在多线程环境下使用。

此外, 还有 Instant 类用于表示时间轴上的时间点, ZoneId 类用于表示时区。联合使用它们可以创建 java.util.Date 对象或者转换 Instant 和本地时间。

```
Date udate = Date.from(LocalDateTime.now().atZone(ZoneId.systemDefault()).toInstant());

Instant instant = Instant.now();
LocalDateTime localDateTime = instant.atZone(ZoneId.systemDefault()).toLocalDateTime();
```

## 8) 注解。

Java 8 中的 Annotations 是可重复的, 即将相同的注解在同一类型上使用多次。

在注解声明时使用 @Repeatable, 可以使同一个注解类型同时使用多次:

```
@interface TestAnnotations {
    TestAnnotation[] value();
}

@Repeatable(TestAnnotations.class)
```

```

@interface TestAnnotation {
    String value();
}

@TestAnnotation("1")
@TestAnnotation("2")
class User{

}

```

使用 `@Repeatable` 标注注解, Java 编译器会隐式地在该注解使用中加入 `@TestAnnotations`。

这样就可以通过反射获取类的注解信息, 但不能直接通过 `TestAnnotation` 获取, 需要使用 `TestAnnotations` 或者用 Java 8 新增加的 `getAnnotationsByType` 方法:

```

User.class.getAnnotation(TestAnnotations.class);
User.class.getAnnotationsByType(TestAnnotation.class);

```

此外, Java 8 中注解的 `@Target` 的使用范围会扩展到两种新的类型。

- `TYPE_PARAMETER`: 注解能写在类型变量的声明语句中。
- `TYPE_USE`: 注解能写在使用类型的任何语句中。

定义一个注解, 将其 `target` 设置为 `ElementType.TYPE_PARAMETER` 或 `ElementType.TYPE_USE`, 或者两个都包含时, 那么此注解就成为类型注解。其可以写在使用类型的任何地方, 主要供开发工具、编译器在编译期做一些检查、转换等工作。如下:

```

@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
public @interface TypeAnnotationsTest {
}

```

```

@TypeAnnotationsTest LocalDateTime dateTime = LocalDateTime.now();

```

### 9) CompletableFuture。

类似于 Guava 提供的 `ListenableFuture`, Java 8 提供了 `CompletableFuture` 简化异步编程的复杂性, 提供了函数式编程的能力, 支持流式调用。如下:

```

static int cal(int loop){
    ...// 计算
    return ...;
}

```

```

CompletableFuture.supplyAsync(() ->
    cal(10))
    .thenCompose((i) -> CompletableFuture.supplyAsync(() -> cal(i)))
    .thenApply((i) -> Integer.toString(i))
    .thenApply((str) -> "result : " + str)

```



```
.thenAccept(System.out::println)
.get();
```

- `supplyAsync`: 提交一个异步任务。
- `thenApply`: 异步任务完成后的回调。
- `get`: 等待整个异步任务完成。

需要注意的是，如果不指定具体的线程池（以 `Async` 结尾并且没有指定 `Executor` 的一些方法），那么 `CompletableFuture` 和并行 `Stream` 一样，都使用的是 `ForkJoin` 的 `commonPool`。

此外，`CompletableFuture` 相比之前的 `Future` 多了一个 `complete` 方法，可以指定完成的时间点，主动触发任务的完成。

Java 8 在 JVM 内存方面也有如下一些改动。

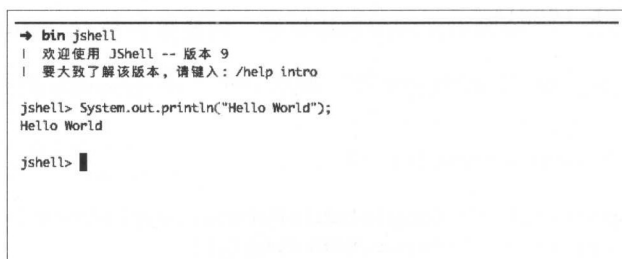
- 取消了方法区（永久代），使用“元空间”替代，元空间只与系统内存相关，也可以通过 `MaxMetaspaceSize` 设置，防止无限耗尽系统内存。
- Java 8 update 20 引入 G1 回收器中的字符串去重（`String deduplication`）。使得 G1 回收器能识别出堆中那些重复出现的字符串并将它们指向同一个内部的 `char[]` 数组，以避免同一个字符串的多份副本，那样堆的使用效率会变得很低。可以使用 `-XX:+UseStringDeduplication` 这个 JVM 参数开启这个特性。

### 7.5.3 Java 9

Java 9 是 Java 中的一个版本，已于 2017.09.21 发布生产可用版本，其带来的新特性主要如下。

#### 1) JShell。

Java 9 带来了许多编程语言都具有的 REPL、JShell，极大地方便了 Java 语言的学习、调试等，如图 7-22 所示。



```
➔ bin jshell
| 欢迎使用 JShell -- 版本 9
| 要大致了解该版本，请键入: /help intro

jshell> System.out.println("Hello World");
Hello World

jshell> █
```

图 7-22

## 2) 模块化。

Java 9 带来的最大变化。Jigsaw 在 Java 7 的时候被移除，在 Java 9 中又回归了。其主要目的是为更小的设备提供可伸缩性，改进 JDK 和 Java SE 的安全性，提升大型应用的性能，并使其更易于构建。

需要注意的是，此特性只是模块化 JDK 源代码，不会改变 JRE 和 JDK 的真实结构。其允许开发者根据项目的需要自定义组件，使用 jlink 工具创建一个只包含所需模块的最小运行时环境，从而减少 rt.jar 的大小，使得 Java 更加容易地应用到小型计算设备中。

## 3) 轻量级的 JSON API。

目前有很多第三方的 JSON API，如 Google 的 Gson、阿里的 FastJson。Java 9 将 JSON API 集成到了 JDK 的实现中。

## 4) 简化了的进程 API。

Java 9 会新增一些进程相关的 API 来增强管理本地系统进程的能力，使得不依赖调用外部程序等变通方案就能够方便灵活地获取本地进程信息、操作进程，提升与操作系统的交互能力。

## 5) 提升访问对象时的线程竞争处理。

线程竞争的锁机制限制了许多 Java 应用的性能。Java 9 改善了锁的争用机制。

## 6) 代码分段缓存。

Java 9 的 JIT 以分段的形式缓存代码。这样能够有更短的扫描时间和更少的碎片，从而提供更好的性能；GC 扫描垃圾时可以直接跳过永驻代码，从而提升效率。

## 7) 用于更大项目构建的智能 Java 编译工具。

Java 9 提供了 Smart Java Compiler，也叫 sjavac，在多核处理器情况下提升 JDK 的编译速度，可用于更大项目的构建。最终目的是取代 javac 成为 Java 环境默认的编译器。

## 8) HTTP/2 客户端。

Java 9 将重新实现一个 HTTP 客户端，支持 HTTP 2.0 和 WebSocket，旨在取代现在的 HttpURLConnection，以解决其自身的不少问题。不过此特性在 Java 9 中标注为 Incubator 版本（并不是最终的 API，后面可能会有大的改动或者被删除）。

## 9) 接口私有方法。

Java 8 给接口引入了默认方法和静态方法，Java 9 进一步引入了接口的私有方法，进一步提高了可重用性。如下：

```
public interface ITest{
    private void test(){
        ...
    }
}
```

#### 10) 响应式编程。

Java 9 引入了新的 Api: `java.util.concurrent.Flow`, 支持响应式的发布 / 订阅框架。`Flow` 类包括以下几个接口。

- `Flow.Publisher`: 消息、事件的生产者。
- `Flow.Subscriber`: 消息、事件的订阅者 / 接受者。
- `Flow.Subscription`: 连接生产者和订阅者的消息控制链路。
- `Flow.Processor`: 可以兼任生产者和订阅者的组件。

#### 11) 集合工厂方法。

和 Guava 中的 `List`、`Set` 等类似, 提供了一系列集合方法:

```
Set.of(1,2,3);
List.of("a","b");
```

#### 12) Stream API。

`Stream` 接口新增了如下几个方法。

- `dropWhile`: 丢弃元素直到第一个不匹配的元素, 即返回去掉匹配断言的最长前缀元素集合后的 `Stream`。
- `takeWhile`: 接收元素直到第一个不匹配的元素, 即返回匹配断言的最长前缀元素集合构成的 `Stream`。
- `ofNullable`: 返回包含一个元素的 `Stream`。
- `iterate`: 根据 `seed` (种子)、`next` (产生下一个元素的方法)、`hasNext` (是否继续产生元素) 参数迭代生成有序 `Stream`。

`Optional` 也加入了 `stream` 方法, 使得其可以作为 `Stream` 进行处理:

```
Optional.of(1).stream()
```

由于 Java 9 发布不久, 应用得并不够广泛, 故在此不对其具体使用进行详述。

## 7.6 总结

本章主要讲述了 Java 开发中的一些高级特性,包括内存管理、网络编程、并发编程,并介绍了常用的 Java 工具库以及 Java 7、8、9 中一些值得使用的新特性。

了解并掌握这些知识,能够使您在构建 Java 应用时尝试使用更高级的技能,从而提升编码效率和编码质量。

除此之外,在平时的 Java 开发中,还有一些容易被忽视的点也需要大家了解。

- float 和 double 只能用来做科学计算或者工程计算,在商业计算中我们要用 `java.math.BigDecimal`。但是如果使用 `BigDecimal(double val)` 构造方法,那么由于小数的 double 底层存储的是一个不确定的数字,使得构造的 `BigDecimal` 也是一个不确定的数字,应该使用 `BigDecimal(String val)` 构造方法做精确计算。
- 在使用基于数组的集合时,如 `ArrayList`、`HashMap` 时,必须指定初始化大小,否则大小不足时,会成倍扩容。
- `String` 自带的 `split` 方法是基于正则表达式的,应尽量避免使用。
- `DateFormat` 类以及其子类是非线程安全的,在多线程环境下不能使用单例。
- 能够避免使用正则表达式的地方应尽量避免使用正则表达式。正则运算对 CPU 的消耗是非常大的,而且会在某些偶然场景下触发死循环正则运算。
- JSON 的序列化和反序列化也都非常消耗 CPU,除非必须得用,则应尽量避免使用,尤其在为了打印类的表示信息时。
- 对于所有外部调用以及内部服务调用都要做容错和超时处理。不管是 RPC 调用还是对于第三方服务的调用,都不能想当然地认为可用性是 100% 的。不允许出现服务调用超时和重试,否则将会对应用程序的稳定性和性能造成不利的影响。

# 第 8 章

## 性能调优

在实际的开发工作中，有时候会遇到程序突然变得响应缓慢或者进程消失的情况，这时候就需要对程序进行问题排查和调优，找出产生问题的根源并进行优化。

一般来说，影响程序性能的因素主要有以下几个。

- 硬件配置。
- 操作系统。
- 应用程序。

对于硬件配置，无须多言，CPU、内存、硬盘等硬件配置从根本上决定了应用的性能。本章暂且抛开这方面不讲，主要介绍针对后两者的调优方法。这两种调优方法又可以分为如下这几种。

- **系统调优**：针对操作系统的配置调优。
- **Java 调优**：针对 Java 应用的调优，包括 JVM 和代码。
- **外部系统调优**：针对应用程序依赖的外部系统，如数据库、缓存、Web 服务器等的调优。

无论针对哪一个层面的调优，性能调优都可以分为以下 3 个步骤。

- **性能监控**：此步骤可以描述为“我并不知道我要做什么”，在系统没有出现问题以前，是没有调优的动机和需求的（当然凭借经验预判除外）。这时需要监控机制来发现、暴露系统的性能问题。这里一般依赖系统级别或者业务级别的监控工具。

- **性能分析**：此步骤可以描述为“我知道我要做什么”，当性能监控发现问题的时候，调优的动机就到了。这时你就知道要去解决什么样的问题了，带着这个问题去做各个层面的分析。此步骤需要一些性能分析工具。
- **性能调优**：此步骤可以描述为“我知道我需要知道什么了”，经过性能分析，你最终会知道是什么原因造成了问题，需要怎么做才能使性能提高。这一步通常需要系统、程序参数的调整，代码的重构优化等。

外部系统的调优在前面的各个章节中已经做了一些阐述，本章主要讲述系统调优和Java调优。

## 8.1 调优准备

调优是需要做好准备工作的，毕竟每一个应用的业务目标都不尽相同，性能瓶颈也不会总在同一个点上。在业务应用层面，我们需要做到以下这几点。

- 需要了解系统的总体架构，明确压力方向。比如系统的哪一个接口、模块是使用率最高的，面临高并发的挑战。
- 需要构建测试环境来测试应用的性能，使用AB、LoadRunner、JMeter、Faban都可以。
- 对关键业务数据量进行分析，这里主要指的是对一些数据的量化分析，如数据库一天的数据量有多少、缓存的数据量有多大等。
- 了解系统的响应速度、吞吐量、TPS、QPS等指标需求，比如对秒杀系统的相应速度和QPS的要求是非常高的。
- 了解系统相关软件的版本、模式和参数等，有时候限于应用依赖服务的版本、模式等，性能也会受到一定的影响。

此外，还需要做以下几个准备工作。

- 了解Java内存相关知识：这一部分在7.1节已经介绍过。
- 对Java代码进行基准性能测试：可以使用JMH来进行，其是一个微基准测试框架，可以进行性能测试，能够去除JIT热点代码编译对性能的影响。
- 了解HotSpot虚拟机体系结构。
- 对系统性能进行调优。
- 了解JVM关键参数的配置。

### 8.1.1 HotSpot 虚拟机体系结构

HotSpot 虚拟机是 Java 开发用得最多的 JVM，了解其体系结构后，才能够从原理层面更好地理解性能问题，也才能更好地对 Java 应用进行调优。

HotSpot VM 主要由垃圾回收器、JIT 编译器以及 HotSpot VM Runtime 组成，如图 8-1 所示。

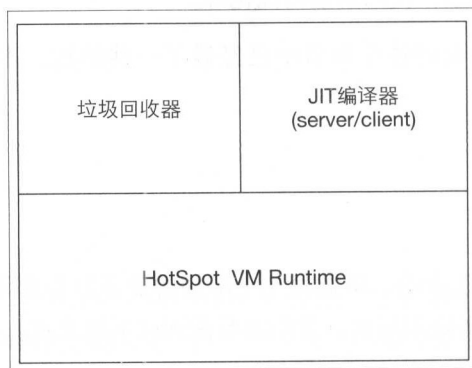


图 8-1

HotSpot VM 的运行时架构包括类加载器、执行引擎以及运行时数据区。如图 8-2 所示，Java 源码被编译器编译为 JVM 字节码后进入 JVM，由类加载器进行加载，并交给执行引擎执行，期间的数据都放入运行时数据区。

需要注意的是，JIT 编译器是执行引擎中非常影响应用性能的组件，它会把热点代码直接编译为本地机器码，从而提高运行时的性能。此外，垃圾回收器执行 GC 的时机、效率对应用性能的影响也非常关键。

HotSpot VM 内部有一些线程进行 JVM 的管理、监控、垃圾回收工作，其主要包括如下这几个。

- **VM thread**: 这个线程是 JVM 里面的线程母体，是一个单例对象（最原始的线程），会产生或触发所有其他的线程，这一单个的 VM 线程是会被其他线程所使用来做一些 VM 操作的（如清扫垃圾等）。
- **Periodic task thread**: 该线程是 JVM 周期性任务调度的线程，它由 WatcherThread 创建，是一个单例对象。
- **Garbage collection threads**: 进行垃圾回收的线程。
- **JIT compiler threads**: 进行 JIT 编译的线程。

- Signal dispatcher thread: 当外部 jvm 命令接收成功后, 会交给 Signal dispatcher 线程并分发到各个不同的模块以处理命令, 之后返回处理结果。

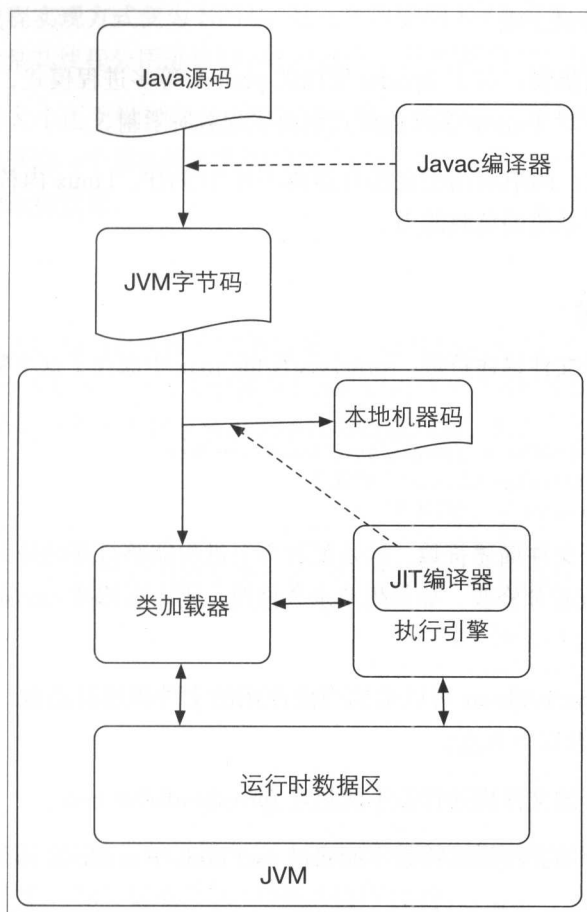


图 8-2

这些线程的运行有时候会影响业务线程的运行, 是影响应用性能的关键因素。

### 8.1.2 操作系统的性能调优

后端应用都是需要部署在服务器上的, 因此在对 Java 应用调优之前务必先将系统的性能调整到一个相对较好的水平。

一般来说, 目前后端系统都是部署在 Linux 主机上的。所以抛开 Windows 系统不谈, 对于 Linux 系统来说一般有以下配置关系着系统的性能。



- **文件描述符数限制：**Linux 中所有东西都是文件，一个 Socket 就对应着一个文件描述符，因此系统配置的最大打开文件数以及单个进程能够打开的最大文件数就决定了 Socket 的数目上限。
- **进程 / 线程数限制：**对于 Apache 使用的 prefork 等多进程模式，其负载能力由进程数目所限制。对 Tomcat 多线程模式则由线程数所限制。
- **TCP 内核参数：**网络应用的底层自然离不开 TCP/IP，Linux 内核有一些与此相关的配置也决定了系统的负载能力。

## 文件描述符数限制

- **系统最大打开文件描述符数：**/proc/sys/fs/file-max 中保存了这个数目，可修改此值。

临时性

```
echo 1000000 > /proc/sys/fs/file-max
```

永久性：在 /etc/sysctl.conf 中设置

```
fs.file-max = 1000000
```

- **进程最大打开文件描述符数：**这是配置单个进程能够打开的最大文件数目，可以通过 ulimit -n 查看和修改。如果想要永久修改，则需要修改 /etc/security/limits.conf 中的 nofile 选项。

通过读取 /proc/sys/fs/file-nr 可以看到当前使用的文件描述符总数。另外，对于文件描述符的配置，需要注意以下几点。

- 所有进程打开的文件描述符数不能超过 /proc/sys/fs/file-max。
- 单个进程打开的文件描述符数不能超过 user limit 中 nofile 的 soft limit。
- nofile 的 soft limit 不能超过其 hard limit。
- nofile 的 hard limit 不能超过 /proc/sys/fs/nr\_open。

## 进程 / 线程数限制

- **进程数限制：**ulimit -u 可以查看 / 修改单个用户能够打开的最大进程数。/etc/security/limits.conf 中的 nproc 则是系统的最大进程数。
- **线程数限制：**
  - 可以通过 /proc/sys/kernel/thread-max 查看系统总共可以打开的最大线程数。
  - 单个进程的最大线程数和 PTHREAD\_THREADS\_MAX 有关，此限制可以在 /usr/include/bits/local\_lim.h 中查看，但是如果想要修改的话，需要重新编译 Linux 系统。

- Linux 内核 2.4 的线程实现方式为 Linux threads，是轻量级进程，会首先创建一个管理线程，线程数目的大小是受 PTHREAD\_THREADS\_MAX 影响的。Linux 2.6 内核的线程实现方式变为 NPTL，是一个改进的 LWP 实现，与 Linux thread 最大的区别就是其线程公用进程的 pid (tgid)，线程数目大小只受制于资源。
- 线程数的大小还受线程栈大小的制约：使用 ulimit -s 可以查看 / 修改线程栈的大小，即每开启一个新的线程需要分配给此线程的一部分内存。减小此值可增加可以打开的线程数目。

## TCP 内核参数

在一台服务器 CPU 和内存资源有限的情况下，最大程度地压榨服务器的性能，是最终目的。在节省成本的情况下，可以考虑修改 Linux 的内核 TCP/IP 参数，来最大程度地压榨服务器的性能。如果通过修改内核参数也无法解决负载问题，那么只能考虑升级服务器了。如下：

```
netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a}]'
```

使用上面的命令，可以得到当前系统的各个状态的网络连接数目。如下：

```
LAST_ACK 13
SYN_RECV 468
ESTABLISHED 90
FIN_WAIT1 259
FIN_WAIT2 40
CLOSING 34
TIME_WAIT 28322
```

这里，TIME\_WAIT 的连接数需要注意。此值过高会占用大量连接，影响系统的负载能力。这时需要调整参数，以尽快地释放 TIME\_WAIT 连接。

一般 TCP 相关的内核参数在 /etc/sysctl.conf 文件中。为了能够尽快释放 TIME\_WAIT 状态的连接，可以做以下配置。

- net.ipv4.tcp\_syncookies = 1，表示开启 SYN Cookies。当出现 SYN 等待队列溢出时，启用 Cookies 来处理，可防范少量 SYN 攻击，默认值是 0，表示关闭。
- net.ipv4.tcp\_tw\_reuse = 1，表示开启复用。允许将 TIME-WAIT Sockets 重新用于新的 TCP 连接，默认值是 0，表示关闭。
- net.ipv4.tcp\_tw\_recycle = 1，表示开启 TCP 连接中 TIME-WAIT Socket 的快速回收，默认值是 0，表示关闭。
- net.ipv4.tcp\_fin\_timeout = 30，修改系统默认的 TIMEOUT 时间。

需要注意的一点是，当打开了 `tcp_tw_recycle` 时，就会检查时间戳，移动环境下发来的包的时间戳有时候是乱跳的，会把带了“倒退”的时间戳的包当作“recycle 的 tw 连接的重传数据，而不是新的请求”，于是丢掉不回包，这样会造成大量丢包。另外，当前面有 LVS，并且采用的是 NAT 机制时，开启 `tcp_tw_recycle` 也会造成一些异常。如果这种情况下仍然需要开启此选项，那么可以考虑设置 `net.ipv4.tcp_timestamps=0`，忽略报文的时间戳即可。

此外，还可以通过优化 TCP/IP 的可使用端口的范围，进一步提升负载能力。如下：

- `net.ipv4.tcp_keepalive_time = 1200`，表示当 Keep Alive 启用的时候，TCP 发送 Keep Alive 消息的频度。默认值是 2 小时，可改为 20 分钟。
- `net.ipv4.ip_local_port_range = 10000 65000`，表示用于向外连接的端口范围。默认情况下很小：32768<sub>61000</sub>，可改为 10000<sub>65000</sub>。这里需要注意不要将最低值设得太低，否则可能会占用正常的端口。
- `net.ipv4.tcp_max_syn_backlog = 8192`，表示 SYN 队列的长度，默认值是 1024，而加大队列长度为 8192 可以容纳更多等待连接的网络连接数。
- `net.ipv4.tcp_max_tw_buckets = 5000`，表示系统同时保持 TIME\_WAIT 的最大数量，如果超过这个数字，TIME\_WAIT 会被立刻清除并打印警告信息。默认值是 180000，可改为 5000，这样可以很好地减少 TIME\_WAIT 套接字数量。

### 8.1.3 系统常用诊断工具

当应用运行情况、响应情况异常时，会直接表现为系统的指标异常，而指标需要通过相关的系统命令来获取。Linux 系统下常用诊断工具如下。

#### 1) uptime

使用 `uptime` 可以快速查看服务器的负载情况：

```
00:40:16 up 116 days, 5:28, 1 user, load average: 0.36, 0.32, 0.32
```

此命令返回的是系统的平均负荷，包括 1 分钟、5 分钟、15 分钟内可以运行的任务平均数量，包括正在运行的任务以及虽然可以运行但正在等待某个处理器空闲的任务。当然，这个值是和 CPU 核数有关的，双核的机器，load 小于 2 也是正常的状况。CPU 的情况可以通过查看 `/proc/cpu` 来获得。

如果 1 分钟平均负载很高，而 15 分钟平均负载很低，说明服务器正在面临高负载情况，需要进一步排查 CPU 资源都消耗在了哪里。反之，如果 15 分钟平均负载很高，1 分钟平均负载较低，则有可能是 CPU 资源紧张时刻已经过去。

## 2) dmesg | tail

该命令会输出系统日志的最后 10 行。常见的 OOM kill 和 TCP 丢包在这里都会有记录:

```
e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
ADDRCONF(NETDEV_UP): eth0: link is not ready
ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
e1000: eth1 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
eth0: no IPv6 routers present
eth1: no IPv6 routers present
e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
e1000: eth1 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
eth0: no IPv6 routers present
eth1: no IPv6 routers present
```

## 3) vmstat 1

vmstat 是一个实时性能检测工具, 可以展现给定时间间隔服务器的状态值, 包括服务器的 CPU 使用率、内存使用、虚拟内存交换情况、I/O 读写情况等系统核心指标。其输出结果如下:

```
procs -----memory----- ---swap-- -----io----- --system-- -----
cpu-----

 r b swpd free buff cache si so bi bo in cs us sy id wa st

0 0 0 7887984 1320604 6288252 0 0 0 2 0 1 0 0 100 0 0
```

一般主要关注输出的 CPU 使用情况, 其中  $id + us + sy = 100$ ,  $id$  是空闲 CPU 使用率,  $us$  是用户 CPU 使用率,  $sy$  是系统 CPU 使用率。如果用户时间和系统时间相加非常大, 说明 CPU 正忙于执行指令。而如果 I/O 等待时间很长, 那么系统的瓶颈可能在磁盘 I/O。当然, 这里输出的 I/O 信息、上下文切换信息也很有用。

在计算 CPU 利用率的时候, 建议多获取几次, 尤其是在脚本里获取时, 一般只获取一次是不准确的, 建议在脚本里取两次以上并排除掉第一次的数据。

## 4) mpstat -P ALL 1

该命令用来显示每个 CPU 的使用情况。如果有一个 CPU 占用率特别高, 说明有可能是一个单线程应用程序引起的。如下:

Linux 2.6.18-194.el5 (xx) 08/01/2017

12:54:59 AM	CPU	%user	%nice	%sys	%iowait	%irq	%soft	%steal
%idle	intr/s							
12:55:00 AM	all	4.74	0.00	1.37	0.00	0.25	1.25	0.00
92.39	3593.07							
12:55:00 AM	0	2.97	0.00	0.99	0.00	0.00	0.00	0.00

```

96.04      992.08
12:55:00 AM    1    0.00    0.00    0.00    0.00    0.00    0.00    0.00
100.00      0.00
12:55:00 AM    2    1.96    0.00    0.98    0.00    0.00    0.98    0.00
96.08      0.00
12:55:00 AM    3    2.00    0.00    1.00    0.00    0.00    0.00    0.00
97.00      0.00
12:55:00 AM    4    2.00    0.00    0.00    0.00    0.00    0.00    0.00
98.00      0.00
12:55:00 AM    5    2.02    0.00    1.01    0.00    0.00    0.00    0.00
96.97      0.99
12:55:00 AM    6    4.00    0.00    0.00    0.00    0.00    0.00    0.00
96.00      0.00
12:55:00 AM    7   23.00    0.00    7.00    0.00    3.00    9.00    0.00
58.00     2599.01

```

#### 5) free -m

该命令可以查看系统内存的使用情况，-m 参数表示按照兆字节展示。如果可用内存非常少，系统可能会动用交换区（swap），这时会增加 I/O 开销（可以在 iostat 命令中体现），降低系统性能：

```

              total          used          free        shared    buffers     cached
Mem:          16051        15879           171            0          672        4763
-/+ buffers/cache:      10444        5607
Swap:           8001            0          8000

```

需要注意的是，第 1 行的信息是针对整个系统来说的，因此 Buffer 和 Cache 都被计算在了 used 里面，其实这两部分内存是可以被很快拿来供应用程序使用的。因此，真正反映内存使用状况的是第 2 行。

#### 6) sar -n DEV 1

sar 命令主要用来查看网络设备的吞吐率。可以通过网络设备的吞吐率，判断网络设备是否已经饱和。如下：

```

Linux 2.6.18-194.el5 (test-172-16-0-137-ip)    08/01/2017

01:06:51 AM      IFACE  rxpck/s   txpck/s   rxbyt/s   txbyt/s   rxcmp/s
txcmp/s  rxmcsst/s
01:06:52 AM      lo      0.00      0.00      0.00      0.00      0.00
0.00      0.00
01:06:52 AM     eth0    1162.00   1286.00  223033.00 323209.00    0.00
0.00      0.00
01:06:52 AM     eth1      1.00      0.00     92.00      0.00      0.00
0.00      0.00
01:06:52 AM     sit0      0.00      0.00      0.00      0.00      0.00
0.00      0.00

```

```

Average:          IFACE  rxpck/s   txpck/s   rxbyt/s   txbyt/s   rxcmp/s
txcmp/s  rxmcast/s
Average:          lo      0.00      0.00      0.00      0.00      0.00
0.00      0.00
Average:          eth0    1162.00   1286.00  223033.00 323209.00      0.00
0.00      0.00
Average:          eth1      1.00      0.00     92.00      0.00      0.00
0.00      0.00
Average:          sit0      0.00      0.00      0.00      0.00      0.00
0.00      0.00

```

### 7) top

`top` 命令包含了系统全局的很多指标信息, 包括系统负载情况、系统内存使用情况、系统 CPU 使用情况等, 基本涵盖了上述几条命令的功能。如下:

```

top - 01:02:04 up 116 days,  5:50,  1 user,  load average: 1.20, 0.46,
0.29
Tasks: 152 total,   2 running, 150 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.7%us,   0.5%sy,   0.0%ni, 96.3%id,   0.0%wa,   0.1%hi,   0.4%si,
0.0%st
Mem: 16436664k total, 16302392k used,   134272k free,   688420k buffers
Swap: 8193140k total,    132k used,   8193008k free,   4878348k cached

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27852	root	20	0	180m	36m	548	R	56.5	0.2	0:00.29	python
4259	root	18	0	2396m	815m	13m	S	25.3	5.1	7988:19	java
14128	root	18	0	4603m	1.9g	15m	S	3.9	12.4	572:23.01	java
14785	root	24	0	5554m	2.1g	25m	S	1.9	13.2	36:42.81	java
27851	root	15	0	12744	1048	748	R	1.9	0.0	0:00.02	top
1	root	15	0	10356	684	576	S	0.0	0.0	0:06.18	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:08.14	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.17	ksoftirqd/0
4	root	RT	-5	0	0	0	S	0.0	0.0	0:08.73	migration/1
5	root	34	19	0	0	0	S	0.0	0.0	0:00.17	ksoftirqd/1

通过此命令, 可以相对全面地查看系统负载的来源。同时, `top` 命令支持排序, 可以按照不同的列排序, 方便地查找出诸如内存占用最多的进程、CPU 占用率最高的进程等。但 `top` 命令是一个瞬时输出的值, 最好通过定时存储其值到文件中来进行对比诊断。

## 8.1.4 JDK 常用诊断工具

在对 Java 程序进行问题排查、性能调优时, 如果没有合适的工具, 很多时候会事倍功半, 甚至无法继续进行下去。其实 JDK 自身已经提供了很多强大的工具供我们使用。

首先, 笔者的开发环境是 OS X El Capitan 10.11.6。

JDK 版本:

```
java version "1.8.0_92"  
Java(TM) SE Runtime Environment (build 1.8.0_92b14)  
Java HotSpot(TM) 64Bit Server VM (build 25.92b14, mixed mode)
```

JAVA\_HOME/bin 下的工具截图如图 8-3 所示。

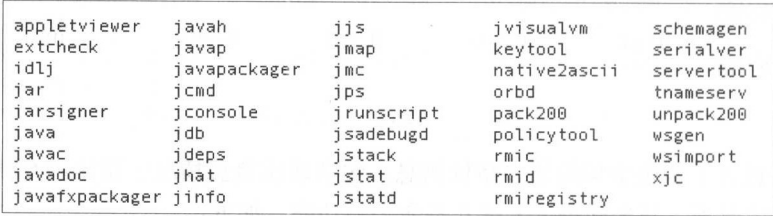


图 8-3

和性能分析相关的工具如表 8-1 所示。

表 8-1

工具	描述
javap	Java 反编译工具，主要用于根据 Java 字节码文件反汇编为 Java 源代码文件
jcmd	Java 命令行（Java Command），用于向正在运行的 JVM 发送诊断命令请求
jconsole	图形化用户界面的监测工具，主要用于监测并显示运行于 Java 平台上的应用程序的性能和资源占用等信息
jdeps	用于分析 Java class 的依赖关系
jdb	Java 调试工具（Java Debugger），主要用于对 Java 应用进行断点调试
jhat	Java 堆分析工具（Java Heap Analysis Tool），用于分析 Java 堆内存中的对象信息
jinfo	Java 配置信息工具（Java Configuration Information），用于打印指定 Java 进程、核心文件或远程调试服务器的配置信息
jmap	Java 内存映射工具（Java Memory Map），主要用于打印指定 Java 进程、核心文件或远程调试服务器的共享对象内存映射或堆内存细节
jmc	Java 任务控制工具（Java Mission Control），主要用于 HotSpot JVM 的生产时间监测、分析、诊断。开发者可以使用 jmc 命令来创建 JMC 工具
jps	JVM 进程状态工具（JVM Process Status Tool），用于显示目标系统上的 HotSpot JVM 的 Java 进程信息
jrunscript	Java 命令行脚本外壳工具（Command Line Script Shell），主要用于解释执行 JavaScript、Groovy、Ruby 等脚本语言
jstack	Java 堆栈跟踪工具，主要用于打印指定 Java 进程、核心文件或远程调试服务器的 Java 线程的堆栈跟踪信息
jstat	JVM 统计监测工具（JVM Statistics Monitoring Tool），主要用于监测并显示 JVM 的性能统计信息，包括 GC 统计信息

续表

工具	描述
jstatd	jstatd（VM jstatd Daemon）工具是一个 RMI 服务器应用，用于监测 HotSpot JVM 的创建和终止，并提供了一个接口，允许远程监测工具附加到运行于本地主机的 JVM 上
jvisualvm	JVM 监测、故障排除、分析工具，主要以图形化界面的方式提供运行于指定虚拟机的 Java 应用程序的详细信息

更多常用的 Java 调优命令可见附录 E。

8.2 性能分析

在系统层面能够影响应用性能的一般包括 3 个因素：CPU、内存和 I/O，可以从这 3 方面进行程序的性能瓶颈分析。

8.2.1 CPU 分析

当程序响应变慢的时候，首先使用 top、vmstat、ps 等命令查看系统的 CPU 使用率是否有异常，从而可以判断出是否是 CPU 繁忙造成的性能问题。其中，主要通过 us（用户进程所占的百分比）这个数据来看异常的进程信息。当 us 接近 100% 甚至更高时，可以确定是 CPU 繁忙造成的响应缓慢。一般说来，CPU 繁忙的原因有以下几个。

- 线程中有无限空循环、无阻塞、正则匹配或者单纯的计算。
- 发生了频繁的 GC。
- 多线程的上下文切换。

确定 CPU 使用率最高的进程之后就可以使用 jstack 来打印异常进程的堆栈信息，如图 8-4 所示。

```
jstack [pid]
```

需要注意的一点是，Linux 下所有线程最终还是以轻量级进程的形式存在于系统中的，而使用 jstack 只能打印进程的信息，这些信息里面包含了此进程下面的所有线程（轻量级进程，LWP）的堆栈信息。因此，需要进一步确定是哪一个线程耗费了大量 CPU 资源，此时可以使用 top -p [processId] -H 来查看，也可以直接通过 ps -Le 来显示所有进程，包括 LWP 的资源耗费信息。最后，通过在 jstack 的输出文件中查找对应的 LWP 的十六进制 ID（printf %0x [processId]），即可定位到相应的堆栈信息。其中需要注意的是，线程的状态



是 RUNNABLE、WAITING 等。对于 RUNNABLE 的进程需要注意是否有耗费 CPU 资源的计算，对于 WAITING 的线程一般是锁的等待操作。

```
Full thread dump Java HotSpot(TM) 64-Bit Server VM (23.5-b02 mixed mode):

"Attach Listener" daemon prio=10 tid=0x0000000011d65000 nid=0x2e78 waiting on condition [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

"pool-3-thread-100" prio=10 tid=0x00000000125e0000 nid=0x153c waiting on condition [0x00002aaade09e000]
  java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x000000007688bd7e8> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)
    at java.util.concurrent.ArrayBlockingQueue.take(ArrayBlockingQueue.java:374)
    at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1043)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1103)
    at java.lang.Thread.run(Thread.java:745)
```

图 8-4

此外，使用 jstack 查看线程栈时需要注意：JVM 只能在 Safepoint 转储出一个线程的栈；由于 jstack dump 实现机制每次只能转储出一个线程的栈信息，因此输出信息中可能会看到一些冲突信息，如一个线程正在等待的锁并没有被其他线程持有，多个线程持有同一个锁等。

也可以使用 jstat 来查看对应进程的 GC 信息，以判断是否是 GC 造成了 CPU 繁忙，如图 8-5 所示。

jstat -gcutil [pid]

S0	S1	E	O	P	YGC	YGCT	FGL	FGLT	GC
0.00	42.25	36.75	53.86	12.24	113	8.258	3	1.755	10.013

图 8-5

还可以使用 vmstat，通过观察内核状态的上下文切换（cs）次数，来判断是否是上下文切换造成的 CPU 繁忙，如图 8-6 所示。

vmstat 1 5

procs		memory				swap		io		system		cpu			
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
1	0	0	5301272	552096	3946624	0	0	0	5	0	0	4	4	92	0
1	0	0	5306356	552096	3946624	0	0	0	84	1034	281	12	1	87	0
1	0	0	5306232	552096	3946624	0	0	0	0	1013	277	12	1	87	0
1	0	0	5306356	552096	3946624	0	0	0	16	1031	282	12	1	87	0

图 8-6

此外，有时候可能会由 JIT 引起一些 CPU 飆高的情形，如大量方法编译等。这里可以使用 -XX:+PrintCompilation 这个参数输出 JIT 编译情况，以排查 JIT 编译引起的 CPU 问题。

## 8.2.2 内存分析

对 Java 应用来说,内存主要是由堆外内存和堆内内存组成的。

### 1) 堆外内存

堆外内存主要是 JNI、Deflater/Inflater、DirectByteBuffer (NIO 中会用到) 使用的。对于这种堆外内存的分析,还需要先通过 vmstat、sar、top、pidstat 等命令查看 Swap 和物理内存的消耗状况再做判断。对于 JNI、Deflater 这种调用可以通过 Google-preftools 来追踪资源使用状况。

### 2) 堆内内存

此部分内存为 Java 应用主要的内存区域。通常与这部分内存性能相关的有如下这几个。

- **创建的对象**: 一般存储在堆中,需要控制好对象的数量和大小,尤其是大对象很容易进入老年代。
- **全局集合**: 全局集合通常生命周期比较长,因此需要特别注意全局集合的使用。
- **缓存**: 缓存选用的数据结构不同,会在很大程度上影响内存的大小和 GC。
- **ClassLoader**: 主要是动态加载类容易造成永久代内存不足。
- **多线程**: 线程分配会占用本地内存,过多的线程也会造成内存不足。

以上使用不当很容易造成:

- 频繁 GC → Stop the world, 使你的应用响应变慢。
- OOM, 直接造成内存溢出错误使得程序退出。OOM 又可以分为以下几种。
  - Heap space: 堆内存不足。
  - PermGen space: 永久代内存不足。
  - Native thread: 本地线程没有足够内存可分配。

排查堆内存问题的常用工具是 jmap, 是 JDK 自带的。一些常用用法如下。

- 查看 JVM 内存使用状况: `jmap -heap <pid>`。
- 查看 JVM 内存存活的对象: `jmap -histo:live <pid>`。
- 把 heap 里所有对象都 dump 下来, 无论对象是死是活: `jmap -dump:format=b, file=xxx.hprof <pid>`。

- 先做一次 Full GC，再 dump，只包含仍然存活的对象信息：jmap -dump:format=b, live, file=xxx.hprof <pid>。

此外，不管是使用 jmap 命令产生的还是在 OOM 时产生的 dump 文件，都可以使用 Eclipse 的 MAT (Memory Analyzer Tool) 来分析，可以看到具体的堆栈和内存中对象的信息。当然 JDK 自带的 jhat 也能够查看 dump 文件，并启动 Web 端口供浏览器浏览。界面如图 8-7 所示。

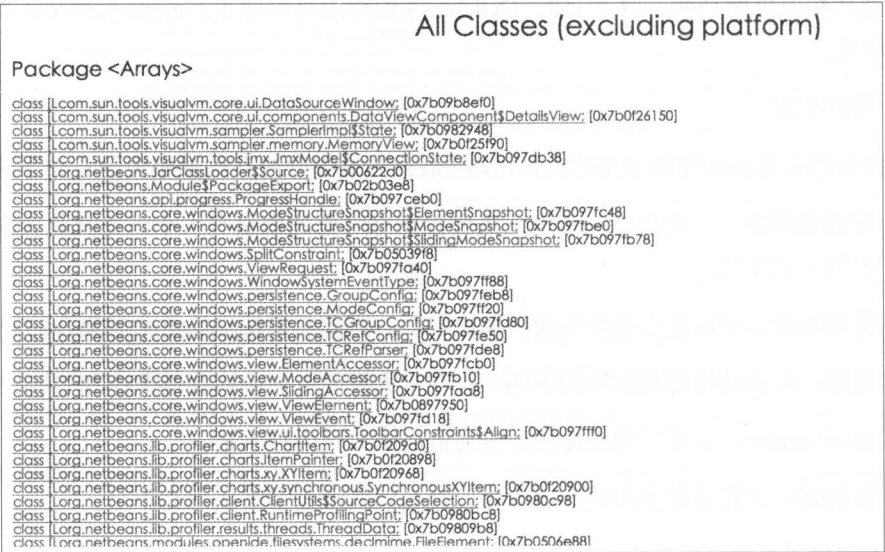


图 8-7

### 8.2.3 I/O 分析

通常与应用性能相关的 I/O 分析包括文件 I/O 和网络 I/O。

#### 1) 文件 I/O。

可以使用系统工具 pidstat、iostat、vmstat 来查看 I/O 的状况。这里可以查看如图 8-8 所示的 vmstat 结果图。

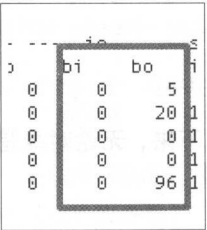


图 8-8

这里主要注意 bi 和 bo 这两个值，分别表示块设备每秒接收的块数量和块设备每秒发送的块数量，由此可以判定 I/O 是否繁忙。然后，可以进一步通过使用 strace 工具定位对文件 I/O 的系统调用。通常，造成文件 I/O 性能差的原因不外乎：

- 大量的随机读 / 写。
- 设备慢。
- 文件太大。

## 2) 网络 I/O。

查看网络 I/O 状况，一般使用的是 netstat 工具。可以查看所有连接的状况、数目、端口信息等。例如，当 TIME\_WAIT 或者 CLOSE\_WAIT 连接过多时，会影响应用的响应速度。如下：

```
netstat -anp
```

netstat 结果图如图 8-9 所示。

tcp	0	0	10.10.151.147:8080	10.10.151.147:39003	TIME_WAIT	-
tcp	0	0	10.10.151.147:50125	10.10.194.156:6379	ESTABLISHED	26015/java
tcp	0	0	10.10.151.147:8081	10.10.164.5:47693	TIME_WAIT	-
tcp	0	0	10.10.151.147:80	10.10.251.30:53402	TIME_WAIT	-
tcp	0	0	10.10.151.147:8081	10.10.164.5:41720	TIME_WAIT	-
tcp	0	0	10.10.151.147:8080	10.10.164.5:45048	TIME_WAIT	-
tcp	0	0	10.10.151.147:49031	10.10.27.133:6379	ESTABLISHED	25112/java
tcp	1	0	10.10.151.147:48313	163.177.71.222:80	CLOSE_WAIT	26015/java
tcp	1	1	10.10.151.147:41063	163.177.71.222:80	LAST_ACK	-
tcp	0	0	10.10.151.147:8080	10.10.151.147:38769	TIME_WAIT	-
tcp	0	0	10.10.151.147:8080	10.10.151.147:49520	TIME_WAIT	-
tcp	1	0	10.10.151.147:45226	163.177.71.222:80	CLOSE_WAIT	25112/java
tcp	0	0	10.10.151.147:8081	10.10.164.5:41644	TIME_WAIT	-
tcp	0	0	10.10.151.147:8080	10.10.151.147:43097	TIME_WAIT	-
tcp	0	0	10.10.151.147:8081	10.10.164.5:52137	TIME_WAIT	-
tcp	0	0	10.10.151.147:44406	10.10.194.156:6379	ESTABLISHED	26015/java
tcp	0	0	10.10.151.147:8081	10.10.151.147:45520	TIME_WAIT	-
tcp	1	0	10.10.151.147:52687	163.177.71.222:80	CLOSE_WAIT	25112/java

图 8-9

此外，还可以使用 tcpdump 来具体分析网络 I/O 的数据。当然，tcpdump 产生的文件直接打开看到的是一堆二进制数据，可以使用 Wireshark 查看具体的连接以及其中数据的内容。如下：

```
tcpdump -i eth0 -w tmp.cap -tnn dst port 8080 # 监听 8080 端口的网络请求并打印日志到 tmp.cap 中
```

还可以通过查看 /proc/interrupts 来获取当前系统使用的中断的情况，如图 8-10 所示。

```
[root@10-10-151-147 ~]# cat /proc/interrupts
```

	CPU0	CPU1	CPU2	CPU3	
0:	128	0	0	0	IO-APIC-edge timer
1:	6	0	0	0	IO-APIC-edge i8042
4:	73	0	0	0	IO-APIC-edge serial
8:	0	0	0	0	IO-APIC-edge rtc0
9:	0	0	0	0	IO-APIC-fasteoi acpi
10:	0	0	0	0	IO-APIC-fasteoi virtio0
11:	27	0	0	0	IO-APIC-fasteoi uhci_hcd:usb1
12:	104	0	0	0	IO-APIC-edge i8042
14:	0	0	0	0	IO-APIC-edge ata_piix
15:	0	0	0	0	IO-APIC-edge ata_piix
24:	0	0	0	0	PCI-MSI-edge virtio3-config
25:	11566222	0	0	0	PCI-MSI-edge virtio3-requests
26:	0	0	0	0	PCI-MSI-edge virtio4-config
27:	232266105	0	0	0	PCI-MSI-edge virtio4-requests
28:	0	0	0	0	PCI-MSI-edge virtio2-config
29:	2	0	0	0	PCI-MSI-edge virtio2-virtqueues
30:	0	0	0	0	PCI-MSI-edge virtio1-config
31:	3561927990	0	0	0	PCI-MSI-edge virtio1-input.0
32:	1427764	0	0	0	PCI-MSI-edge virtio1-output.0
NMI:	0	0	0	0	Non-maskable interrupts
LOC:	3543714566	182648482	188792377	311749419	Local timer interrupts
SPU:	0	0	0	0	Spurious interrupts
PMI:	0	0	0	0	Performance monitoring interrupts
IWI:	0	0	0	0	IRQ work interrupts
RES:	3484622506	8433697	4264869522	4242309417	Rescheduling interrupts
CAL:	124739	1697010350	1827712931	1758918087	Function call interrupts
TLB:	321062994	359440141	357431134	356252082	TLB shutdowns
TRM:	0	0	0	0	Thermal event interrupts
THR:	0	0	0	0	Threshold APIC interrupts
MCE:	0	0	0	0	Machine check exceptions
MCP:	90504	90504	90504	90504	Machine check polls
ERR:	0				
MTS:	0				

图 8-10

各列依次是：

irq 的序号，在各自 CPU 上发生中断的次数，可编程中断控制器，设备名称（request\_irq 的 dev\_name 字段）

通过查看网卡设备的终端情况可以判断网络 I/O 的状况。

## 8.2.4 其他分析工具

上面分别针对 CPU、内存以及 I/O 介绍了一些系统 JDK 自带的分析工具。除此之外，还有一些综合分析工具或者框架可以更加方便我们对 Java 应用性能的排查、分析、定位等。

- VisualVM

这个工具应该是 Java 开发者们非常熟悉的一款 Java 应用监测工具，原理是通过 JMX 接口来连接 JVM 进程，从而能够看到 JVM 上的线程、内存、类等信息。VisualVM 界面如图 8-11 所示。

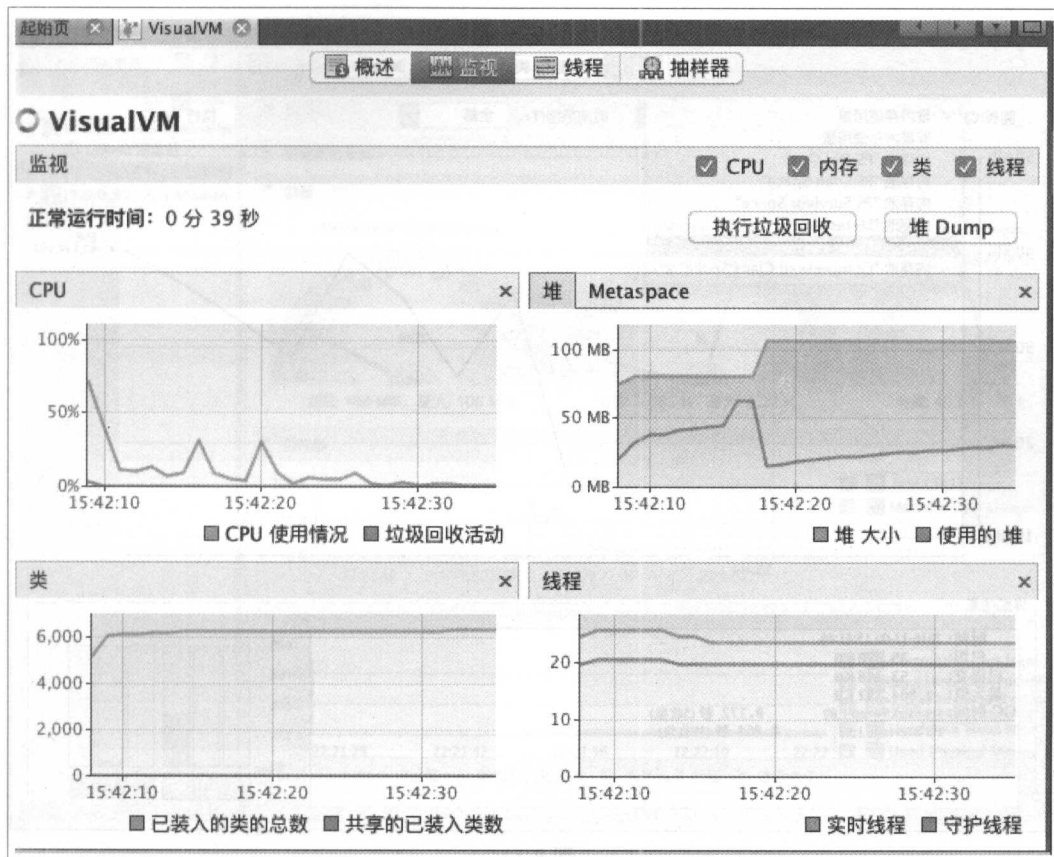


图 8-11

如果想进一步查看 GC 情况，可以安装 Visual GC 插件。此外，VisualVM 也有 Btrace 的插件，可以可视化、直观地编写 Btrace 代码并查看输出日志。与 VisualVM 类似，JConsole 也是通过 JMX 查看远程 JVM 信息的一款工具，更进一步地，通过它还可以显示具体的线程堆栈信息以及内存中各个年代的占用情况，也支持直接远程执行 MBEAN。当然，VisualVM 通过安装 JConsole 插件也可以拥有这些功能。

JConsole 界面如图 8-12 所示。

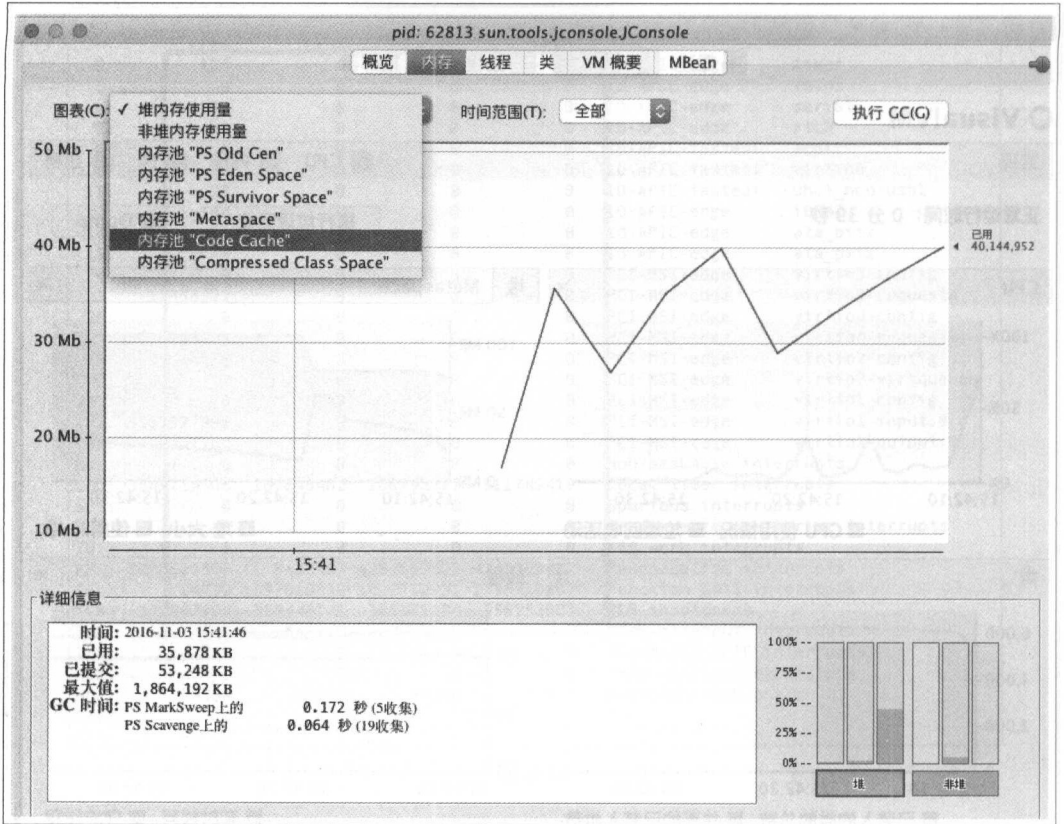


图 8-12

但由于这两个工具都需要 UI 界面，因此一般都是通过本地远程连接服务器 JVM 进程。在服务器环境下，通常并不用这种方式。

- Java Mission Control (JMC)

此工具是 JDK 7 u40 开始自带的，原来是 JRockit 上的工具，是一款采样型的集诊断、分析和监控于一体的非常强大的工具，其界面如图 8-13 所示。但此工具基于 JFR (jcmd JFR.start name=xx duration=60s settings=template.jfc filename=xx.jfr)，而开启 JFR 需要商业证书 (jcmd VM.unlock\_commercial\_features)。

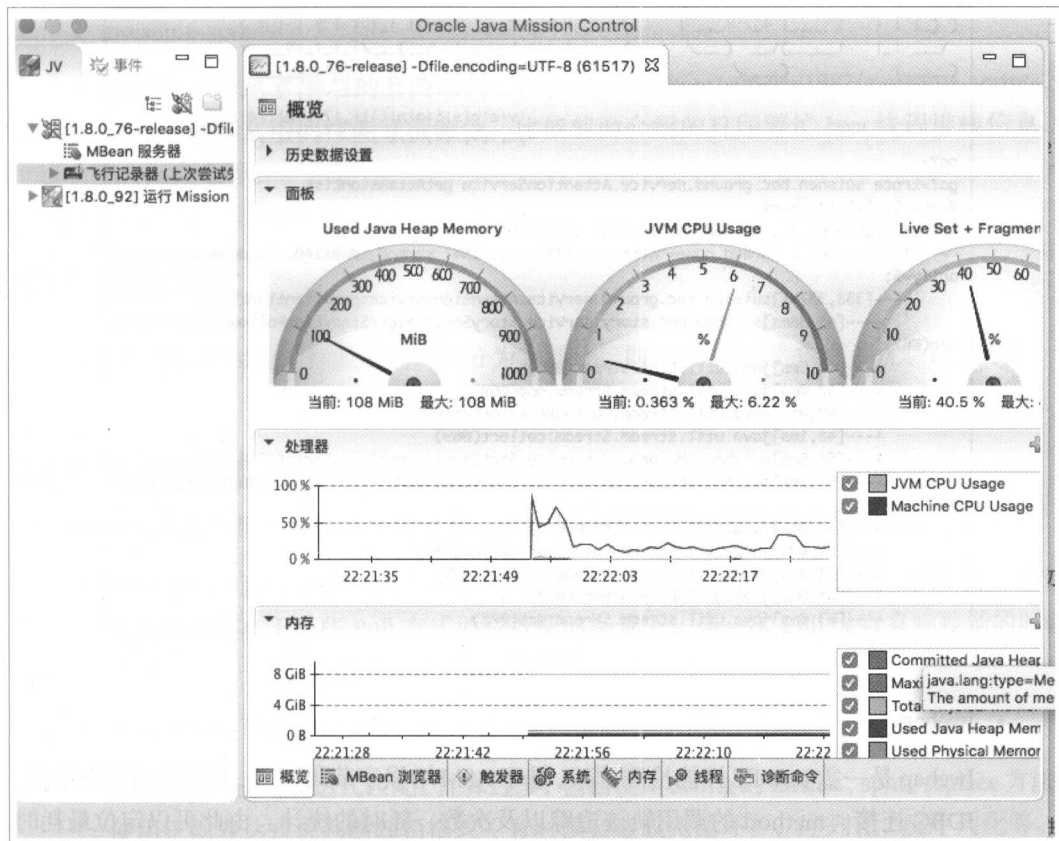


图 8-13

- Btrace 和 Greys

这里不得不提的是 Btrace 这个神器，它使用 Java Attach API + Java Agent + Instrument API 实现了 JVM 的动态追踪。在不重启应用的情况下可以加入拦截类的方法以打印日志等。但是此工具使用时需要自己编写脚本，比较麻烦。推荐使用原理和 Btrace 类似的 Greys: <https://github.com/oldmanpushcart/greys-anatomy>。Greys 的使用示例如图 8-14 所示。



```
( ( ( | | | | | _ | | | | | / _ | | | | | | | | | | | | | | |
  \ _ | | | | | _ ) \ ( \ _ | | | | | \ _ ) \ | | | | | \ _ )
( _ | | | | | _ )

+++++
|v|e|r|i|s|i|o|i|n|:|1|.|17|.|16|.|14|
+++++
```

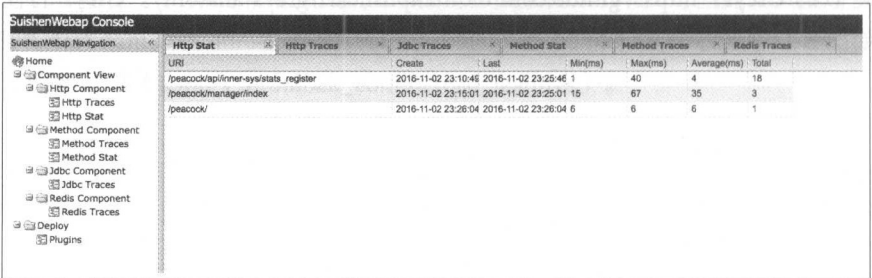
```
ga?>
ga?>trace suishen.bbc.ground.service.AttentionService getAttentionList
Press Ctrl+D to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 773 ms.
`---+Tracing for : thread_name="http-apr-8080-exec-154" thread_id=0x140a;is_daemon=true;priority=5;
`---+ [338,337ms]suishen.bbc.ground.service.AttentionService:getAttentionList()
`---+ [43,38ms]suishen.bbc.story.service.StoryService:getStoryListFollowUserOrCreate
ser(@67)
`---+ [44,0ms]java.util.List:stream(@69)
`---+ [47,0ms]java.util.stream.Stream:map(@69)
`---+ [47,0ms]java.util.stream.Collectors:toList(@69)
`---+ [48,1ms]java.util.stream.Stream:collect(@69)
`---+ [54,6ms]suishen.bbc.ground.service.StatService:getStoryStats(@71)
`---+ [57,3ms]suishen.bbc.ground.service.StoryGroundService:maxGroundNoOfStoryMap(@7:
)
`---+ [57,0ms]java.util.List:stream(@75)
`---+ [61,0ms]java.util.stream.Collectors:toMap(@75)
`---+ [61,0ms]java.util.stream.Stream:collect(@75)
`---+ [61,0ms]java.util.List:stream(@77)
`---+ [63,0ms]java.util.stream.Stream:map(@78)
```

图 8-14

• Jwebap

Jwebap 是一款 Java EE 性能检测框架, 基于 ASM 增强字节码实现, 其支持 HTTP 请求、JDBC 连接、method 的调用轨迹追踪以及次数、耗时的统计。由此可以定位最耗时的请求、方法, 并可以查看 JDBC 连接的次数、是否关闭等。但此项目是 2006 年的一个项目, 已经将近 10 年没有更新。根据笔者使用后了解到, 它已经不支持 JDK 7 编译的应用。如果要使用该工具, 建议基于原项目二次开发, 同时也可以加入对 Redis 连接的轨迹追踪。当然, 基于字节码增强的原理, 也可以实现自己的 Java EE 性能监测框架。

图 8-15 来自笔者公司二次开发过的 Jwebap, 已经支持 JDK 8 和 Redis 的连接追踪。



SuishenWebap Console							
SuishenWebap Navigation		Http Stat	Http Traces	Jdbc Traces	Method Stat	Method Traces	Redis Traces
		URI	Create	Last	Min(ms)	Max(ms)	Average(ms) Total
Home		/peacock/ap/inner-sys/stats_register	2016-11-02 23:10:45	2016-11-02 23:25:46	1	40	4 18
Component View		/peacock/manager/index	2016-11-02 23:15:01	2016-11-02 23:25:01	15	67	35 3
Http Component		/peacock/	2016-11-02 23:28:04	2016-11-02 23:28:04	6	6	1
Http Stat							
Method Component							
Method Traces							
Method Stat							
Jdbc Component							
Jdbc Traces							
Redis Component							
Redis Traces							
Deploy							
Plugins							

图 8-15

- awesome-scripts

这里有一个笔者参与的开源项目：<https://github.com/superhj1987/awesome-scripts>，封装了很多常用的性能分析命令，比如前面介绍的打印繁忙 Java 线程堆栈信息、Greys 命令等。

## 8.3 性能调优

与性能分析相对应，性能调优同样分为 3 部分，即 CPU 调优、内存调优、I/O 调优。

### 8.3.1 CPU 调优

CPU 调优主要是合理安排运算过程，达到充分但不过度使用 CPU 的目的。

- 不要存在一直运行的线程（无限 while 循环），可以使用 Sleep 休眠一段时间。这种情况普遍存在于一些 Pull 方式消费数据的场景下，当一次 Pull 没有拿到数据的时候建议 Sleep 一下，再做下一次 Pull。
- 轮询的时候可以使用 wait/notify 机制。
- 避免循环、正则表达式匹配、计算过多，包括使用 String 的 format、split、replace 方法；使用正则表达式去判断邮箱格式（有时候会造成死循环）；序列化 / 反序列化等。
- 结合 JVM 和代码，避免产生频繁的 GC，尤其是 Full GC。

此外，使用多线程的时候，还需要注意以下几点。

- 使用线程池，减少线程数以及线程的切换。
- 多线程对于锁的竞争可以考虑减小锁的粒度（使用 ReentrantLock）、拆开锁（类似 ConcurrentHashMap 分 bucket 上锁），或者使用 CAS、ThreadLocal、不可变对象等无锁技术。此外，多线程代码的编写最好使用 JDK 提供的并发包、Executors 框架以及 Fork/Join 等，此外 Disruptor 和 Actor 在合适的场景下也可以使用。

### 8.3.2 内存调优

内存的调优主要就是对 JVM 的调优。

- 合理设置各个代的大小。避免新生代设置过小（不够用，经常 Minor GC 并进入老年代）以及过大（会产生碎片），同样也要避免 Survivor 设置过大和过小。

- 选择合适的 GC 策略。需要根据不同的场景选择合适的 GC 策略。这里需要说的是，CMS 并非全能的，除非特别需要再设置。毕竟 CMS 的新生代回收策略 ParNew 并非最快的，且会产生碎片。此外，G1 直到 JDK 8 的出现也没有得到广泛应用，故并不建议使用。
- 老年代优先使用 Parallel GC（-XX:+UseParallel[Old]GC），可以保证最大的吞吐量。由于 CMS 会产生碎片，确实有必要才改成 CMS 或 G1。
- 垃圾回收的最佳状态是只有 Young GC，也就是避免生命周期很长的对象存在。
- 从 Young GC 开始，尽量给新生代大一点的内存，避免 Full GC。
- 注意 Survivor 大小的设置，不能太大也不能太小。
- 注意内存墙（严重阻碍处理器性能发挥的内存瓶颈），一般将单点应用堆内存设置为 4~5GB 即可，依靠可扩展性提高并发能力。
- 设置 JVM 的内存大小有一个经验法则：完成 Full GC 后，应该释放出 70% 的内存。
- 打开 GC 日志并读懂 GC 日志，以便于排查问题。GC 日志文件可以使用 GC Histogram（gchisto）生成图标和表格。

```
-XX:PrintHeapAtGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps  
-XX:+PrintGCTimeStamp -Xloggc:$CATALINA_BASE/logs/gc.log
```

其中，对于上面的第一条，具体还有如下建议。

- **新生代大小选择：**响应时间优先的应用，尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在这种情况下，新生代回收发生 GC 的频率是最低的。同时，也能够减少到达老年代的对象。吞吐量优先的应用，也应尽可能地设置大些，因为对响应时间没有要求，垃圾回收可以并行进行，适合 8 CPU 以上的应用使用。
- **老年代大小选择：**响应时间优先的应用，老年代一般都是使用并发回收器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等参数。如果堆设置小了，会造成内存碎片、高回收频率以及应用暂停而使用传统的标记-整理方式；如果堆设置大了，则会造成较长的 GC 时间。最优化的方案，一般需要参考以下数据获得。
  - 并发垃圾回收信息。
  - 持久代并发回收次数。
  - 传统 GC 信息。
  - 花在新生代和老年代回收上的时间比例。

吞吐量优先的应用应该有一个很大的新生代和一个较小的老年代，这样可以尽可能地回收大部分短期对象，减少中期对象，而老年代存放长期存活对象。

这里还需要着重介绍一下使用并发回收器时较小堆引起的碎片问题。因为老年代的并发回收器使用标记 - 清除算法，所以不会对堆进行压缩。尤其当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发回收器找不到足够的空间，那么并发回收器将会停止，然后使用传统的标记 - 整理方式进行回收。如果出现“碎片”，需要进行如下配置：-XX:+UseCMSCompactAtFullCollection，开启对老年代的压缩（合并相邻空间）。同时使用 -XX:CMSFullGCsBeforeCompaction=xx 设置多少次 Full GC 后，对老年代进行压缩。

其余对于 JVM 的优化问题可见后面 JVM 参数配置部分。

代码上，也需要注意：

- 小对象分配的代价很小，通常 10 个 CPU 指令；新对象也非常廉价；不用担心活得很短的小对象。
- 大对象分配的代价以及初始化的代价很大；不同大小的大对象可能导致 Java 堆碎片，尤其是 CMS、ParallelGC 或 G1；尽量避免分配大对象。
- 避免改变数据结构大小，如避免改变数组或 array backed collections / containers 的大小；对象构建（初始化）时最好显式批量确定数组大小；改变大小导致的不必要的对象分配，可能会引发 Java 堆碎片。
- 避免保存重复的 String 对象，同时也需要小心 String.substring() 与 String.intern() 的使用，中间过程会生成不少字符串。
- 尽量不要使用 finalizer。
- 释放不必要的引用：ThreadLocal 使用完记得释放以防止内存泄漏，各种 stream 使用完也记得 close。
- 使用对象池避免无节制创建对象，造成频繁 GC。但不要随便使用对象池，除非像连接池、线程池这种初始化 / 创建资源消耗较大的场景。对象池可能潜在的问题如下。
  - 增加了活对象的数量，可能增加 GC 时间。
  - 访问（多线程）对象池需要锁，可能带来可扩展性的问题。
  - 小心过于频繁的对象池访问。
- 缓存失效算法，可以考虑使用 SoftReference、WeakReference 保存缓存对象。
- 谨慎热部署 / 加载的使用，尤其是动态加载类等。

- 不要用 Log4j 输出文件名、行号，因为 Log4j 通过打印线程堆栈实现，会生成大量 String。此外，使用 Log4j 时，建议使用这种经典用法，先判断对应级别的日志是否打开，再做操作，否则也会生成大量 String。如下：

```
if (logger.isInfoEnabled()) {  
    logger.info(msg);  
}
```

此外，在部署 Java EE 容器时经常会有一种争论：使用大内存容器好还是多个小的容器集群好？这需要根据业务场景区别对待。通常，大内存容器有以下问题。

- 一旦发生 Full GC，会非常耗时。
- 一旦 GC，dump 出的堆快照太大，无法分析。

因此，如果可以保证程序中的对象大部分都是朝生夕死的，老年代不会发生 GC，那么使用大内存容器是可以的。但是大内存容器在伸缩性和高可用性上却比不上使用小内存（相对来说）容器集群。使用小内存容器集群则有以下优势。

- 可以根据系统的负载调整容器的数量，以达到资源的最大利用率。
- 可以防止单点故障。

### 8.3.3 I/O 调优

文件 I/O 上需要注意：

- 考虑使用异步写入代替同步写入，可以借鉴 Redis 的 AOF 机制。
- 利用缓存，减少随机读。
- 尽量批量写入，减少 I/O 次数和寻址。
- 使用数据库代替文件存储。

网络 I/O 上需要注意：

- 和文件 I/O 类似，使用异步 I/O、多路复用 I/O、事件驱动 I/O 代替同步阻塞 I/O。
- 批量进行网络 I/O，减少 I/O 次数。
- 使用缓存，减少对网络数据的读取。
- 使用协程：Quasar。

### 8.3.4 其他优化建议

其他优化建议如下。

- 算法和逻辑是程序性能的首要优化对象，遇到性能问题，应该首先优化程序的处理逻辑。
- 优先考虑使用返回值而不是异常表示错误。
- 查看自己的代码是否对内联是友好的。

此外，网上有一些过时的建议，读者可以参考。

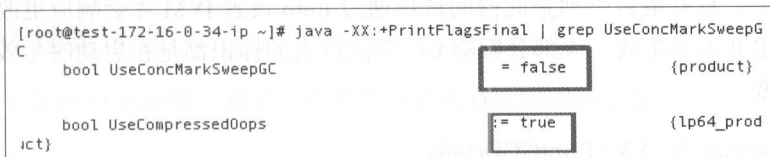
- 变量用完设置为 NULL，可以加快内存回收。这种用法在大部分情况下没有意义，但有一种情况除外：如果有一个 Java 方法没有被 JIT 编译但里面仍然有代码会执行比较长时间，那么在这段会长时间执行的代码前显式地将不需要的引用类型局部变量设置为 NULL 是可取的。
- 方法参数设置为 final，这种用法也没有太大的意义，尤其在 JDK 8 中引入了 effective final，会自动识别 final 变量。

### 8.3.5 JVM 参数配置

JVM 的参数配置在很大程度上影响 Java 应用的性能。而 JVM 参数众多，很容易混淆和设置错误。这里针对 Oracle/Sun JDK 6 介绍一些需要注意的 JVM 参数配置。

#### 1) 启动参数默认值。

Java 有很多的启动参数，而且版本各不一样。但是现在网上充斥着各种资料，如果不加辨别地全部使用，很多是没有效果或者本来就有默认值的。一般地，我们可以通过使用 `java -XX:+PrintFlagsInitial` 来查看所有可以设置的参数及其默认值，也可以在程序启动的时候加入 `-XX:+PrintCommandLineFlags` 来查看与默认值不相同的启动参数。如果想查看所有启动参数（包括和默认值相同的），可以使用 `-XX:+PrintFlagsFinal`，如图 8-16 所示。



```
[root@test-172-16-0-34-ip ~]# java -XX:+PrintFlagsFinal | grep UseConcMarkSweepGC
C
    bool UseConcMarkSweepGC          = false          {product}

    bool UseCompressedOops            := true            {lp64_prod
ict}
```

图 8-16

输出里“=”表示使用的是初始默认值，而“:=”表示使用的不是初始默认值，可能是命令行传进来的参数、配置文件里的参数或者是 Ergonomics（自动优化机制）自动选择了

别的值。最后一列中 `product` 表示所有平台的默认值都一样, `pd_product` 则表示默认值因平台不同而不同。

此外, 还可以使用 `jinfo` 命令显示启动参数:

```
jinfo -flags [pid] # 查看目前启动使用的有效参数
```

```
jinfo -flag [flagName] [pid] # 查看对应参数的值
```

这里需要指出的是, 当你配置 JVM 参数时, 最好是先通过以上命令查看对应参数的默认值, 之后再确定是否需要设置。最好不要配置你搞不清用途的参数, 毕竟默认值的设置自有合理之处。

## 2) 动态设置参数。

在 Java 应用启动后, 发现是 GC 造成的性能问题, 但是你启动的时候并没有加入打印 GC 的参数, 很多时候的做法就是重新加参数然后重启应用, 但这样会造成一定时间的服务不可用。最佳的做法是能够在不重启应用的情况下, 动态设置参数。使用 `jinfo` 可以做到这一点 (本质上还是基于 JMX 的):

```
jinfo -flag [+/-][flagName] [pid] # 启用 / 禁止某个参数
```

```
jinfo -flag [flagName=value] [pid] # 设置某个参数
```

对于上述 GC 的情况, 就可以使用以下命令打开 `HeapDump` 并设置 `dump` 路径。

```
jinfo -flag +HeapDumpBeforeFullGC [pid]
```

```
jinfo -flag +HeapDumpAfterFullGC [pid]
```

```
jinfo -flag HeapDumpPath=/home/dump/dir [pid]
```

同样也可以动态关闭:

```
jinfo -flag -HeapDumpBeforeFullGC [pid]
```

```
jinfo -flag -HeapDumpAfterFullGC [pid]
```

其他的参数设置类似。

这里需要注意的是, 并非所有的参数通过这种方式设置都能生效。1) 中 `PrintFlagsFinal` 信息的最后一列的值除了 `product` 和 `pd_product`, 还会出现 `manageable` (运行时可以动态更改标识的值)。只有最后一列是此值的选项通过 `jinfo` 设置 JVM 才会响应更改, 否则即使设置成功, 也并不会生效。大部分影响 GC 算法行为的标识都是在启动时生效, 无法通过 `jinfo` 动态改变。

## 3) `-verbose:gc` 与 `-XX:+PrintGCDetails`。

很多 GC 推荐设置都同时设置这两个参数, 其实只要打开了 `-XX:+PrintGCDetails`, 前面的选项也会被同时打开, 无须重复设置。

#### 4) -XX:+DisableExplicitGC。

这个参数的作用就是使得 `system.gc` 变为空调用，很多推荐设置里面都建议开启该设置。但是，如果你用到了 NIO 或者其他使用到堆外内存的情况，使用此选项会造成 OOM，这时可以用 `XX:+ExplicitGCInvokesConcurrent` 或 `XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses`（配合 CMS 使用，使得 `system.gc` 触发一次并发 GC）代替。

此外，还有一个比较有意思的地方。如果你不设置此选项，当你使用了 RMI 的时候，会周期性地 Full GC。这个现象是由分布式 GC 造成的。

#### 5) MaxDirectMemorySize。

此参数设置的是堆外内存的上限值。此参数默认值为 -1，此值为 -Xmx 减去一个 survivor space 的预留大小。

#### 6) 由于遗留问题，作用相同的参数有如下两对。

- -Xss 与 -XX:ThreadStackSize。
- -Xmn 与 -XX:NewSize。这里需要注意的是，设置了 -Xmn，NewRatio 就不会发生作用了。

#### 7) -XX:MaxTenuringThreshold。

使用工具查看此值的默认值为 15，但是选择了 CMS 的时候，此值会变成 4。当此值设置为 0 时，所有 Eden 里的活对象在经历第一次 Minor GC 的时候就会直接晋升到老年代，不会使用 Survivor Space。

#### 8) -XX:HeapDumpPath。

使用此参数可以指定 -XX:+HeapDumpBeforeFullGC、-XX:+HeapDumpAfterFullGC、-XX:+HeapDumpOnOutOfMemoryError 触发 HeapDump 时文件的存储位置。

附录 E 的最后一部分给出了一个 Tomcat 的 JVM 参数配置示例。

### 8.3.6 JVM 性能增强

JDK 7、8 在 JVM 的性能上做了一些增强以提高 Java 应用的性能。

#### 1) 多层编译。

通过 -XX:+TieredCompilation 开启 JDK 7 的多层编译。多层编译结合了客户端 C1 编译器和服务器端 C2 编译器的优点（客户端编译能够快速启动和及时优化，服务器端编译可以提供更多的高级优化），是一个非常高效利用资源的切面方案。



在开始时先进行低层次的编译，同时收集信息，在后期再进一步进行高层次的编译并进行高级优化。

需要注意的一点：这个参数会消耗比较多的内存资源，因为同一个方法被编译了多次，存在多份 native 内存副本，建议把代码缓存调大一点儿（`-XX:+ReservedCodeCacheSize`，`InitialCodeCacheSize`），否则有可能由于代码缓存不足，JIT 编译的时候不停地尝试清理代码缓存，丢弃无用方法，消耗大量资源在 JIT 线程上。

## 2) Compressed Oops。

压缩指针在 JDK 7 中的 Server 模式下已经默认开启。

## 3) Zero-Based Compressed Ordinary Object Pointers。

当使用了上述的压缩指针时，在 64 位 JVM 上，会要求操作系统保留从一个虚拟地址 0 开始的内存。如果操作系统支持这种请求，那么就会开启 Zero-Based Compressed Oops。这样无须在 Java 堆的基地址添加任何地址补充即可把一个 32 位对象的偏移解码成 64 位指针。

## 4) 逃逸分析 (Escape Analysis)。

Server 模式的编译器会根据代码的情况，来判断相关对象的逃逸类型，从而决定是否在堆中分配空间，是否进行标量替换（在栈上分配原子类型局部变量）。此外，也可以根据调用情况来决定是否自动消除同步控制，如 `StringBuffer`。这个特性从 Java SE 6u23 开始就已默认开启。

## 5) NUMA Collector Enhancements。

这一特性主要针对的是 The Parallel Scavenger 垃圾回收器，使其能够利用 NUMA 架构机器的优势来更快地进行 GC。可以通过 `-XX:+UseNUMA` 开启支持。

# 第 9 章

## 安全技术

在表面平静的互联网应用下面,其实并不是那么平静。每天都有无数的安全漏洞被爆出,有的是被白帽子黑客发现提醒大家,而有的则早已被人利用做了一些“坏”事情;每天也有无数的安全攻击在进行,或是对竞品进行 DDOS,或是插入一些非安全代码来获取用户数据。

随着攻击现象越来越多,安全问题正越来越受到各个互联网公司的重视,都从各个方面加固自身的安全壁垒,包括硬件层面、软件层面、行政层面等。作为 Java 开发工程师也需要了解并能够掌握常用安全技术的使用,只有这样,才能减少应用被入侵的可能性,从而最大化避免损害公司利益。

本章主要讲述 Java 开发中常用的安全技术以及一些常见 Web 安全问题的解决思路。包括:

- Java 加密。
- HTTPS。
- 常见 Web 安全问题。

### 9.1 Java 加密

当需要加密用户数据、验证请求发起者身份、散列数据的时候,都是需要加解密算法的。JDK 对常用的加解密算法都已经做了支持。包括:

- 单向加密算法。

- 对称加密算法。
- 非对称加密算法。

### 9.1.1 单向加密算法

单向加密算法指的是接收一段明文，然后以一种不可逆的方式将它转换成一段密文，简单地说就是能够加密数据，但是不能够解密回原来的明文的加密方式。常用的算法包括如下 3 种。

- MD5。
- SHA。
- HMAC。

#### MD5

MD5，全称 Message Digest Algorithm 5，信息摘要算法，是计算机安全领域广泛使用的一种散列函数，用以提供消息的完整性保护以及散列数据的功能。虽然现在有彩虹表能够在一定程度上使用碰撞进行解密，但在使用合适的 salt 的情况下，被解密的概率是微乎其微的。

MD5 能够将无论多长的数据最后都编码成 128 位数据，且对同样的数据最后的加密结果会一直保持一致，因此常用于文件校验、密码加密、散列数据等。

JDK 中的实现如下：

```
byte[] data = ...; // 明文数据
MessageDigest md5 = MessageDigest.getInstance("md5");
md5.update(data); // 加密后的数据
```

上面说的加 salt 指的是：对明文进行 MD5 加密后，拼接一个随机字符串（在用于存储用户密码时，可以选择用户的信息作为 salt），然后再进行一次 MD5，如下：

```
密文 = MD5( MD5( 明文 ) + salt )
```

#### SHA

SHA，全称 Secure Hash Algorithm，安全散列算法，被广泛地应用于电子商务等信息安全领域。其用途和 MD5 类似，但是安全性要高于 MD5，且其最终的加密结果是 160 位数据。

其 JDK 代码实现如下：

```
MessageDigest sha = MessageDigest.getInstance("SHA1");
sha.update(data);
```

## HMAC

HMAC, Hash Message Authentication Code, 散列消息鉴别码, 是基于密钥的 hash 算法的认证协议。

HMAC 的认证原理是, 使用一个密钥生成一个固定大小的小数据块, 即 MAC, 并将其加入消息中, 然后传输。接收方利用与发送方共享的密钥进行鉴别认证等。经常用于对 API 参数进行请求验证: 分配给授权调用方一个密钥, 授权调用方使用密钥和接口的相关信息散列计算出请求签名, 然后将签名连同数据一起发给服务方; 服务方根据调用方标识, 使用其密钥和同样的散列算法计算出签名和传来的签名做比较, 以验证请求的合法性。

### 1) 初始化密钥。

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");

SecretKey secretKey = keyGenerator.generateKey();
byte[] secret = secretKey.getEncoded();
```

### 2) HMAC 加密。

```
SecretKey secretKey = new SecretKeySpec(secret, "HmacMD5");
Mac mac = Mac.getInstance(secretKey.getAlgorithm());
mac.init(secretKey);

byte[] data = ...; // 明文
mac.doFinal(data);
```

这里需要注意的是, MAC 算法除了 HmacMD5, 还有:

- HmacSHA1。
- HmacSHA256。
- HmacSHA384。
- HmacSHA512。

## 9.1.2 对称加密算法

对称加密又称被为单密钥加密, 是采用单钥密码系统的加密方法, 同一个密钥既可以加密, 也可以解密。一般的通信加密流程如图 9-1 所示。

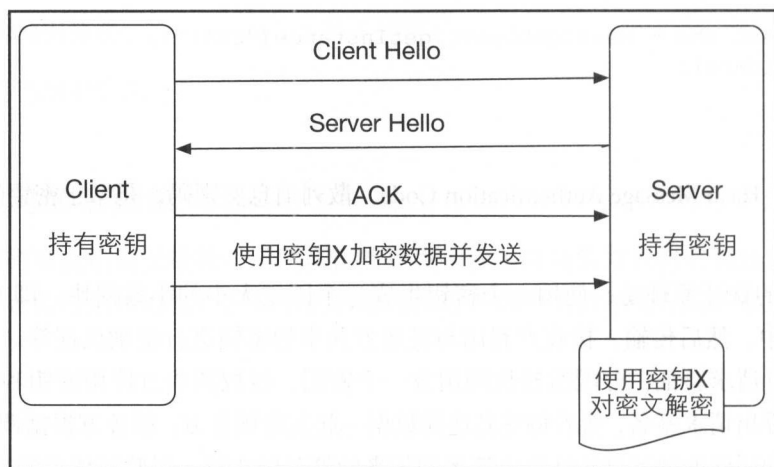


图 9-1

Java 中常用的对称加密算法有 DES、AES 和 PBE，下面分别介绍。

## DES

DES，全称 Data Encryption Standard，即数据加密标准，其使用 64 位的密钥把 64 位的明文输入块经过 16 轮替换和移位操作后变为 64 位的密文输出块。此算法的入口参数有 3 个。

- Key: 8 字节共 64 位，是 DES 算法的工作密钥，但 DES 实际使用其中的 56 位。
- Data: 8 字节共 64 位，是要被加密或解密的数据。
- Mode: DES 的工作方式，包括加密或解密。

加密:

```
byte[] secret = ..; // 密钥
DESKeySpec keySpec = new DESKeySpec(secret);
SecretKey key = SecretKeyFactory.getInstance("DES").
generateSecret(keySpec);

Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] encryData = cipher.doFinal(data); // 加密数据
```

解密:

```
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.DECRYPT_MODE, key);
cipher.doFinal(encryData); // 解密数据
```

## AES

AES 是 DES 的升级版, 相比 DES, 其具有以下特点。

- 运算速度快。
- 对内存的需求非常低。
- 支持可变分组长度, 分组长度可设定为 32bit 的任意倍数, 最小值为 128bit, 最大值为 256bit。
- 密钥长度可设定为 32bit 的任意倍数, 范围为 128~256bit。

加解密代码和 DES 基本一致:

```
byte[] secret = ..; // 密钥
SecretKey key = new SecretKeySpec(secret, "AES");

Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] encryData = cipher.doFinal(data); // 加密数据

Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.DECRYPT_MODE, key);
cipher.doFinal(encryData); // 解密数据
```

## PBE

PBE, 全称 Password-Based Encryption, 基于密码加密, 是一种简便的加密方式。密钥由用户自己掌管, 不借助任何物理媒体, 其采用 salt 杂凑多重加密等方法保证数据的安全性。

### 1) 生成 salt。

```
byte[] salt = new byte[8];
SecureRandom random = new SecureRandom();
random.nextBytes(salt);
```

### 2) 加密。

```
String password = ..; // 用户口令
PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
SecretKey secretKey = keyFactory.generateSecret(keySpec);

PBEParameterSpec paramSpec = new PBEParameterSpec(salt, 200); // 迭代 200 次
Cipher cipher = Cipher.getInstance("PBKDF2WithHmacSHA1");
cipher.init(Cipher.ENCRYPT_MODE, secretKey, paramSpec);

byte[] encryData = cipher.doFinal(data);
```

## 3) 解密。

```

PBEPParameterSpec paramSpec = new PBEPParameterSpec(salt, 200); // 迭代 200 次
Cipher cipher = Cipher.getInstance("PBEWITHMD5andDES");
cipher.init(Cipher.DECRYPT_MODE, secretKey, paramSpec);

cipher.doFinal(encryData);

```

## 9.1.3 非对称加密算法

非对称加密需要两个密钥，一个是公开的，称为公钥；另一个是私有的，称为私钥。公钥用来加密数据，只有私钥才能解密；私钥一般用来签名，公钥用来验证签名。非对称加密算法安全性要比对称加密算法高很多，但是其运算消耗资源多、效率慢，因此很多情况下都是结合对称加密一起使用的。一个简单的非对称加密通信过程如图 9-2 所示。

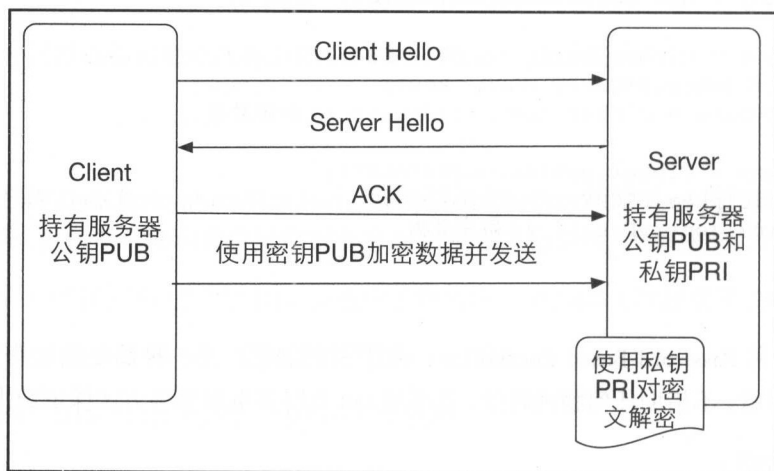


图 9-2

Java 中常用的非对称加密算法有 RSA 和 DH。

## RSA

RSA，是以算法发明者的名字命名的，其安全性依赖于大数分解的困难程度，主要用于认证和数据加解密，也可以用于密钥交换。其一般流程如下。

- 1) A 构建一对密钥，将公钥公布给 B，将私钥保留。
- 2) A 使用私钥加密数据并签名，发送给 B。
- 3) B 使用 A 的公钥、签名来验证收到的密文是否有效，有效则使用 A 的公钥对数据解密。

4) B 使用 A 的公钥加密数据, 发送给 A。A 使用自己的私钥进行解密。

代码示例如下。

1) 生成公钥和私钥。

```
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
keyPairGen.initialize(1024);
```

```
KeyPair keyPair = keyPairGen.generateKeyPair();
```

```
// 公钥
```

```
byte[] publicKey = keyPair.getPublic().getEncoded();
```

```
// 私钥
```

```
byte[] privateKey = keyPair.getPrivate().getEncoded();
```

2) 公钥加密数据。

```
X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(publicKeyBytes);
```

```
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
```

```
Key publicKey = keyFactory.generatePublic(x509KeySpec);
```

```
// 对数据加密
```

```
Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
```

```
cipher.init(Cipher.ENCRYPT_MODE, publicKey);
```

```
String plainText = "plain text.";
```

```
byte[] encryData = cipher.doFinal(plainText.getBytes());
```

3) 私钥解密数据。

```
PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(privateKeyBytes);
```

```
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
```

```
Key privateKey = keyFactory.generatePrivate(pkcs8KeySpec);
```

```
// 对数据解密
```

```
Cipher cipher = Cipher.getInstance(keyFactory.getAlgorithm());
```

```
cipher.init(Cipher.DECRYPT_MODE, privateKey);
```

```
cipher.doFinal(encryData);
```

如上, 私钥使用加密模式, 公钥使用解密模式, 即可实现私钥签名、公钥验证的流程。

## DH

DH, 全称 Diffie-Hellman 算法, 是一个密钥交换协议。其主要用来做密钥交换, 一般不用来做认证和加解密数据。其一般的使用流程如下。

1) A 构建一对密钥, 将公钥公布给 B, 将私钥保留。

2) B 通过 A 公钥构建密钥对儿, 将公钥公布给 A, 将私钥保留。



3) AB 双方互通本地密钥算法。

4) AB 双方公开自己的公钥, 使用对方的公钥和刚才产生的私钥加密数据, 同时可以使用对方的公钥和自己的私钥对数据解密。

可见相比 RSA, DH 要多发送一个 DH 公钥, 并且其最终对数据的加解密依赖于本地对称加密算法。

Java 代码如下。

1) 初始化 A 的公钥和私钥。

```
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("DH");
keyPairGenerator.initialize(1024); // 密钥字节数
KeyPair keyPair = keyPairGenerator.generateKeyPair();

byte[] aPublicKey = keyPair.getPublic().getEncoded(); // A 的公钥
byte[] aPrivateKey = keyPair.getPrivate().getEncoded(); // A 的私钥
```

2) 初始化 B 的公钥和私钥。

```
X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(aPublicKey);
KeyFactory keyFactory = KeyFactory.getInstance("DH");
PublicKey aPubKey = keyFactory.generatePublic(x509KeySpec);

DHParameterSpec dhParamSpec = ((DHPublicKey) aPubKey).getParams();
// 由 A 的公钥构建 B 的密钥对
KeyPairGenerator keyPairGenerator = KeyPairGenerator.
getInstance(keyFactory.getAlgorithm());

keyPairGenerator.initialize(dhParamSpec);

KeyPair keyPair = keyPairGenerator.generateKeyPair();

byte[] bPublicKey = keyPair.getPublic().getEncoded(); // B 的公钥
byte[] bPrivateKey = keyPair.getPrivate().getEncoded(); // B 的私钥
```

3) 用 A 的公钥和 B 的私钥构建密文。

```
String plainText = "你好";

KeyFactory keyFactory = KeyFactory.getInstance("DH");
X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(aPublicKey);
PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);

PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(bPrivateKey);
Key priKey = keyFactory.generatePrivate(pkcs8KeySpec);

KeyAgreement keyAgree = KeyAgreement.getInstance(keyFactory.
getAlgorithm());
keyAgree.init(priKey);
```

```
keyAgree.doPhase(pubKey, true);
```

```
SecretKey secretKey = keyAgree.generateSecret("DES");    // 本地密钥
Cipher cipher = Cipher.getInstance(secretKey.getAlgorithm());
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
```

```
byte[] encryData = cipher.doFinal(plainText.getBytes());
```

DH 需要对称加密算法对数据加密, 这里使用 DES。

4) 用 B 的公钥、A 的私钥解密。

```
KeyFactory keyFactory = KeyFactory.getInstance("DH");
X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(bPublicKey);
PublicKey pubKey = keyFactory.generatePublic(x509KeySpec);

PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(aPrivateKey);
Key priKey = keyFactory.generatePrivate(pkcs8KeySpec);
```

```
KeyAgreement keyAgree = KeyAgreement.getInstance(keyFactory.
getAlgorithm());
keyAgree.init(priKey);
keyAgree.doPhase(pubKey, true);
```

```
SecretKey secretKey = keyAgree.generateSecret("DES");    // 本地密钥
```

```
Cipher cipher = Cipher.getInstance(secretKey.getAlgorithm());
cipher.init(Cipher.DECRYPT_MODE, secretKey);
```

```
cipher.doFinal(encryData);
```

## 9.2 安全 HTTP——HTTPS

很多时候后端应用都会直接提供 HTTP 接口以供浏览器、客户端、第三方来调用。但由于 HTTP 协议是明文传输的, 且没有任何的身份认证机制和数据完整性保证, 因此数据很容易被中间人劫持、监听、篡改等。虽然能够通过对传输的业务数据加密避免这一点, 但 HTTPS 才是最根本的解决方案。这也是现在好多互联网公司都在进行全站 HTTPS 迁移的原因, 也是应用商店要求应用调用的 API 接口都换成 HTTPS 的动机所在。

HTTP 和 HTTPS 的对比如图 9-3 所示。

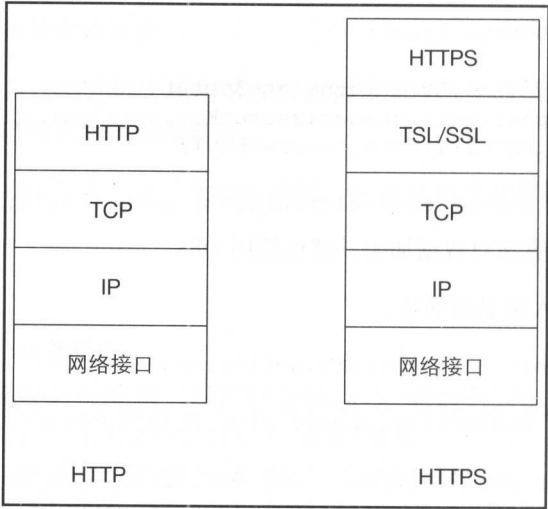


图 9-3

可以看到，HTTPS 相比 HTTP 多了一个安全加密层，不仅对数据进行了加密，还对数据完整性提供了保护，并且也提供了身份验证的功能。

9.1 节最后提到，非对称加密算法在很多情况下会与对称加密算法一起使用，而 HTTPS 就是一个典型的应用场景。简单说就是：其中一方先生成一个对称加密密钥，然后通过非对称加密的方式来发送这个密钥，这样双方之后的通信就可以用对称加密这种高效率的算法进行加解密了。

9.2.1 安全协议——SSL/TLS

SSL 和 TLS 都是用于保障端到端之间连接的安全性的，位于应用层和传输层之间。SSL 现在已经改名为 TLS，主流版本为 TLS 1.2，其结构如图 9-4 所示。

- 握手层：端与端之间协商密码、连接状态等连接参数，并完成身份验证。
- 记录层：对数据的封装，将数据交给传输层之前会经过分片、压缩、认证、加密等操作。

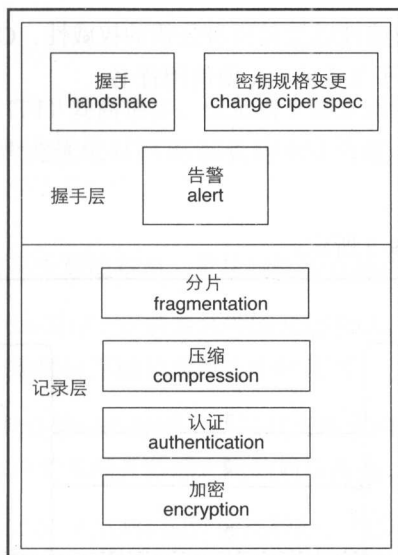


图 9-4

### 9.2.2 证书中心——CA

CA 是 HTTPS 依赖的关键组件，即 Certificate Authority，证书中心。客户端从 CA 获取服务器公钥，并能够保证此公钥不会被中间人篡改。

CA 对公钥完整性保证依赖的一个机制就是颁发证书，证书包括以下内容。

- 证书的发布机构。
- 证书的有效期。
- 公钥。
- 证书所有人。
- 数字签名。

使用 CA 的流程如下。

1) 首先将公钥与个人信息用 hash 算法生成一个消息摘要，然后 CA 再用它的私钥对消息摘要加密，最终形成数字签名。

2) 客户端接收到证书时，用同样的 hash 算法再次生成一个消息摘要，然后用 CA 的公钥对证书进行解密，之后再对比两个消息摘要即可保证数据未被篡改。

此外，为了保证 CA 的权威性以及其自身公钥的权威性，CA 机构有一个树形结构，父节点是信用高的 CA，它会对子节点的 CA 做信用背书。

## 9.2.3 请求交互过程

HTTPS 的交互过程如图 9-5 所示。

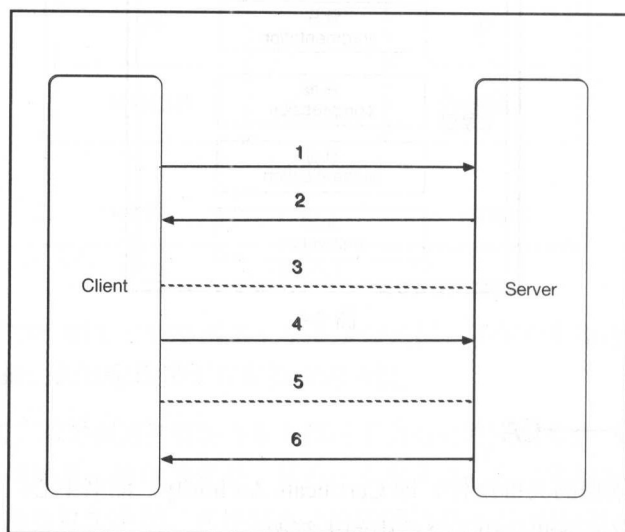


图 9-5

1) 客户端向服务器端发送请求，将客户端的功能和首选项传送给服务器端，包括客户端支持的 SSL 版本、加密组件列表等。

2) 服务器端发送选择的连接参数（从客户端加密组件中筛选出的加密组件内容和压缩方法）以及证书（包含公钥等信息）给客户端。

3) 客户端读取证书中的所有人、有效期等信息并进行校验，然后通过预置的 CA 验证证书合法性，有问题则提示。

4) 客户端生成用于数据加密的对称密钥，然后用服务器的公钥进行加密并发送给服务器端。

5) 服务器端使用自己的私钥解密数据，获得用于数据加密的对称密钥。

6) 安全的通道建立完毕，后续基于对称加密算法传输数据。

## 9.2.4 性能优化

虽然 HTTPS 安全性比 HTTP 要高很多,但是由于建立通信通道要先交互很多次,应用的性能受到了不小的影响,因此优化 HTTPS 的性能非常关键。

### 1) 算法选择。

HTTPS 的通信过程中有不少算法参与,算法的性能直接决定了 HTTPS 的性能。

- **数字签名:** 选择 ECDSA 算法,它的签名性能远超 RSA,而且签名是在服务器端做的。服务器端发送给客户端的证书链包含所有中间证书。
- **密钥交换:** ECDHE 具有更好的性能,并且其支持前向保密 (Forward Secrecy),可以避免中间人保存客户端和服务器端之间的通信数据,并且能开启 TLS False Start。
- **对称加密:** AES256-GCM-SHA384 的性能比较好,建议选择此算法进行数据加密。

### 2) TLS 缓冲区。

TLS 缓冲区大小即一个 TLS Record 的大小,在 Nginx 中默认值是 16KB。如果 HTTP 的数据是 320KB,那么就会被拆分为 20 个 TLS Record,然后每个 TLS Record 会被 TCP 层拆分为多个 TCP 包发送给客户端。

如果此值过小,那么 TLS Record Head 的负载就会增加,会降低连接的吞吐量;而如果此值过大,拆分出的 TCP 包就比较大,传输过程中容易出现丢包,整个 TLS Record 到达客户端的时间就会加长。

由于在 TCP 慢启动的过程中 TCP 连接的拥塞窗口 CWND 较小, TCP 连接吞吐量也小,因此可以把 TLS Record Size 设置得小一点;而在 TCP 连接结束慢启动之后,吞吐量上来了, TLS Record Size 可以设置得大一些。

### 3) TLS False Start。

主要指的是客户端这边的 TLS False Start。开启此选项,那么客户端在发送 Change Cipher Spec、Finished 之后,可以立即发送应用数据,无须等待服务器端的 Change Cipher Spec、Finished。这样,应用数据的发送实际上并未等到握手全部完成,从而节省出一个 RTT 时间,可以提高一定的性能。

但开启此选项,需要满足以下条件。

- 客户端和服务端都需要支持 NPN/ALPN (浏览器要求)。
- 需要采用支持前向保密的密码套件 (ECDHE)。

#### 4) Session Cache 和 Session Ticket。

服务器端对于一次 Session 是有缓存的。如果能够在它的 Session Cache 中找到对应 Session ID 的 session-state (存储协商好的密码套件等信息), 那么服务器端就不必再经历一系列 Session 建立过程。因此, 开启 Session Cache 是提升连接性能的有效措施之一。

此外, 服务器端可以通过某种机制将 session-state 加密后作为 Ticket 发送给客户端。客户端凭借该 Ticket 就可以恢复先前的会话。这就是 Session Ticket 机制。开启此机制也能够简化 Session 建立过程, 提高性能。

以上优化措施都是运维层面的, 对于 Java 开发来说, 可以通过在应用层做预连接, 在网页端或者客户端用户发起访问请求之前提前完成这个握手过程, 从而降低延迟, 这样能在一定程度上提高连接的性能。

## 9.3 Web 安全

Java 开发的一个主要应用场景就是 Web, 即使不是 Web, 很多时候也会采用和 Web 类似的处理方式。因此了解目前常见的 Web 安全问题并做防范是非常关键的。

Web 安全问题, 从大的方面可以分为:

- **客户端安全:** 通过浏览器进行攻击的安全问题。
- **服务器端安全:** 通过发送请求到服务器端进行攻击的安全问题。

常见的客户端安全问题有:

- 跨站点脚本攻击。
- 跨站点请求伪造。

常见的服务器端安全问题有:

- SQL 注入。
- 基于约束条件的 SQL 攻击。
- DDOS 攻击。
- Session fixation。

本节主要针对这些问题进行介绍。

### 9.3.1 跨站点脚本攻击

跨站点脚本攻击, 全称 Cross Site Script (XSS), 顾名思义是跨越两个站点的攻击方式。一般指的是攻击方通过“HTML”注入的方式篡改了网页, 插入了恶意的脚本, 从而在用户浏览网页或者移动客户端使用 WebView 加载时, 默默地做了一些控制操作。

XSS 可以说是客户端安全的首要问题, 稍有不注意就会暴露出相关接口被人利用。

一个 XSS 攻击的例子如下。

- 一个 Java 应用提供了一个接口可以上传个人动态, 动态内容是富文本的。
- 攻击者上传的内容如下:

```

```

- 在服务器端和客户端程序未做任何过滤的情况下, 当其他用户访问这个动态页面时, 就会执行这个脚本。

如果脚本不是一个 alert, 而是换成跳转到一个具有删除操作的 URL, 或者脚本获取用户的 Cookie, 然后发送到远程服务器上, 那么危害就会非常大。

防范这种攻击的常用方式有以下几种。

- 对任何允许用户输入的地方做检查, 防止其提交脚本相关特殊字符串, 如 script、onload、onerror 等。客户端和服务端都要做检查。
- 做输入过滤, 即将特殊字符都过滤掉或者换成 HTML 转义后的字符。在 Java 中可以使用 Apache Commons-Lang 中 StringEscapeUtils 的带 escape 前缀的方法来做转义。
- 给 Cookie 属性设置上 HttpOnly, 可以防止脚本获取 Cookie。
- 对输出内容做过滤。这个可以在客户端做, 也可在服务器端做。服务器端主要就是转义 HTML 字符, 客户端可以使用 escape 方法来过滤。

### 9.3.2 跨站点请求伪造

跨站点请求伪造, 全称 Cross Site Request Forgery, 简称 CSRF。这也是一种常见的攻击方式。

这种攻击方式, 主要通过诱导用户单击某些链接, 从而隐含地发起对其他站点的请求, 进而进行数据操作。



一个攻击示例如下。

- 一个用户登录了一个站点，访问 `http://xx/delete_notes?id=xx` 即可删除一个笔记。
- 攻击者在它的站点中构造一个页面，HTML 页面含有以下内容：

```

```

- 当用户被诱导访问攻击者的站点时就发起了一个删除笔记的请求。

对于 CSRF 攻击的常用解决方案有以下几种。

- 对重要请求要求输入验证码，这样就能防止在用户不知情的情况下，被发送请求。
- 使用类似防盗链的机制，对 header 的 refer 进行检验以确认请求来自合法的源。
- 对重要请求都附带一个服务器端生成的随机 Token，提交时对此 Token 进行验证。这也是业界一个很普遍的做法。

### 9.3.3 SQL 注入攻击

SQL 注入攻击是一个很常见的攻击方式，原理是通过发送特殊的参数，拼接服务器端的 SQL 字符串，从而达到改变 SQL 功能的目的。

一个攻击例子如下。

- 服务器端登录验证使用下面的方式，其中 `userName` 和 `userPwd` 都是用户直接上传的参数。

```
String sql = "select * from user where user_name = '" + userName + "'  
and pwd = " + userPwd;
```

- 用户提交 `userName` 为 `admin'`，`userPwd` 为字符串 `xxx`（由用户确定）。
- 拼接好之后的 SQL 语句变成了：`select * from user where user_name = 'admmmin'-'and pwd = 'xxx'`（-- 为 SQL 语句的注释），这样只要存在 `user_name` 为 `admin` 的用户，此语句就能成功执行并返回 `admin` 用户的信息。

需要说明的是，如果服务器的请求错误信息没有做进一步封装，直接把原始的数据库错误返回，那么有经验的攻击者通过返回结果多次尝试就会有找出 SQL 注入的机会。

防范这种攻击的方案有以下几种。

- 在 Java 中构造 SQL 查询语句时，杜绝拼接用户参数，尤其是拼接 SQL 查询的 `where` 条件。全部使用 `PreparedStatement` 预编译语句，通过 `?` 来传递参数。

- 在业务层面, 过滤、转义 SQL 特殊字符, Apache Commons-Lang 中的 StringEscapeUtil 提供了 escapeSQL 功能 (最新的 Lang 3 已经删除此方法, 因为其只是简单地替换 ' 为 ")。

### 9.3.4 基于约束条件的 SQL 攻击

基于约束条件的 SQL 攻击的原理如下。

- 在处理 SQL 中的字符串时, 字符串末尾的空格字符都会被删除, 包括 WHERE 子句和 INSERT 语句, 但 LIKE 子句除外。
- 在任意 INSERT 查询中, SQL 会根据 varchar(n) 来限制字符串的最大长度, 即超过  $n$  个字符的字符串只保留前  $n$  个字符。

如此, 我们设计一个用户表 (暂且忽略设计的合理性), 对其中的用户名和密码字段都设置 25 个字符限制:

```
CREATE TABLE test_user (
  `user_name` varchar(25),
  `pwd` varchar(25)
);
```

有一个 user\_name 为 user\_test 的用户注册, 于是向数据库添加一条记录:

```
insert into test_user values("user_test", "111111");
```

接着, 一个 user\_name 为 “user\_test 1” (中间留有 25 个空格) 的用户再来注册。一般的业务逻辑如下。

- 判断用户名是否存在。

```
select * from test_user where user_name = 'user_test 1'
```

因为查询语句不会截断字符串, 因此这样获取不到记录, 表示用户不存在。

- 用户名不存在, 那么插入新用户。

```
insert into test_user values("user_test 1", "123456")
```

这样, 由于 user\_name 限制为 25 个字符, 那么新用户的 user\_name 成为了 “user\_test” (后面是 16 个空格字符)。现在数据库记录如下 (第二个记录后面是 16 个空格):

user_name	pwd
user_test	111111
user_test	123456

这样, 当使用 `user_name='user_test'` 和 `pwd='123456'` 登录时, 能匹配到第二条记录, 登录是成功的。但是用户信息使用的是第一条记录, 于是攻击者就获取到了第一个用户的操作权限。

防范这种攻击的措施如下。

- 为具有唯一性的那些列添加 UNIQUE 索引。
- 在数据库操作前先将输入参数修剪为特定长度。

### 9.3.5 分布式拒绝服务攻击——DDOS

DDOS, 全称 Distributed Denial of Service, 分布式拒绝服务攻击。攻击者利用很多台机器同时向某个服务发送大量请求, 人为构造并发压力, 从而使得服务被冲垮, 无法为正常用户提供服务。常见的 DDOS 攻击包括:

- SYN flood。
- UDP flood。
- ICMP flood。

其中 SYN flood 是最经典的 DDOS 攻击。其利用了 TCP 连接三次握手时需要先发送 SYN 的机制, 通过发送大量 SYN 包使得服务器端建立大量半连接, 这就消耗了非常多的 CPU 资源和内存。针对这种攻击, 很多解决方案是在 TCP 层就使用相关算法识别异常流量, 直接拒绝建立连接。但是, 如果攻击者控制很多机器对一个资源消耗比较大的服务接口发起正常访问请求, 那么这个方式就无效了。

由于难以区分是否是正常用户的请求, 因此 DDOS 是非常难以防范的, 但仍有一些措施能够尽量地减少 DDOS 带来的影响, 介绍如下。

- 合理使用缓存、异步等措施提高应用性能。应用抗并发的能力越强, 就越不容易被 DDOS 冲垮服务。
  - 合理使用云计算相关组件, 自动识别高峰流量并做自动扩容。
  - 在应用中限制来自某一 IP 或者某一设备 ID 的请求频率。超过此频率就将其放入黑名单, 下次请求直接拒绝服务。Java 中可以通过 Redis 的 `incr` 和 `expire` 操作来达到。
- 如下:

```
String ip = NetworkUtil.getClientIP(request, false); // 获取客户端 IP 地址
String key = "ddos." + ip;
```

```

long count = suishenRedisTemplate.incr(key); //incr 不会影响 expire
if (count > 10000) {
    throw new AccessException("access too frequently with ip: "
        + StringUtils.defaultString(ip));
} else {
    if (count == 1) {
        suishenRedisTemplate.expire(key, 10);
    }
    return true;
}

```

上述代码即可将同一 IP 的请求限制在 10 秒 10000 次。

此逻辑越靠近访问链路的前面效果越好，比如直接在 Nginx 中拦截，其效果就要比在业务应用中做得好。

### 9.3.6 会话固定攻击——Session fixation

Session fixation 攻击，顾名思义就是会话固定攻击。在我们平时的 Web 开发中都是基于 Session 做用户会话管理的。在浏览器中，Session 的 ID 一般是存储在 Cookie 中的，甚至直接附带在 query 参数中。如果 Session 在未登录变为登录的情况下不发生改变的话，Session fixation 攻击就形成了。

一个攻击示例如下。

- 攻击者进入网站 `http://xx.com`。
- 攻击者发送 `http://xx.com?JSESSIONID=123456` 给一个用户。
- 用户单击此链接进入网站，由于 URL 后面有 JSESSIONID，因此直接使用它作为 Session 的 ID。
- 用户成功登录后，攻击者就可以利用伪造的 Session ID 获取用户的各种操作权限。

这种攻击的关键点就在于 Tomcat 使用 JSESSIONID 作为 Session ID。因此，防范这种攻击的核心之一就在于，不能使用客户端传来的 Session ID。此外还有以下方法。

- 不要接受由 GET 或者 POST 参数指定的 Session ID 值。
- 针对每一个请求都生成新的 Session。
- 只接受服务器端生成的 Session ID。
- 为 Session 指定过期时间。

在 Java Web 项目中，可以实现一个拦截器，将使用 query 参数传递 JSESSIONID 的请求的 Session 删除：

```
public void doFilter(ServletRequest request, ServletResponse response,
                    FilterChain chain) throws IOException,
ServletException
{
    ...

    if (httpRequest.isRequestedSessionIdFromURL()) {
        HttpSession session = httpRequest.getSession();
        if (session != null) {
            session.invalidate();
        }
    }
    ...
}
```

此外，对于每一次登录后的 Session 都重新生成 ID，并设置合理的失效期：

```
public JsonResult login(@RequestBody LoginRequestBody requestBody,
                       HttpServletRequest request)
{
    ...
    boolean loginResult = doLogin();
    if (loginResult) {
        request.changeSessionId(); // 重新生成 Session ID
        request.getSession().setMaxInactiveInterval(1800); // 30 分钟失效
    }
    ...
}
```

# 附录 A

## 代码构建常用命令

### A.1 Maven 常用命令

- 清除。

```
mvn clean
```

- 打包。

```
mvn package
```

- 发布到本地。

```
mvn install
```

- 发布到线上。

```
mvn deploy
```

- 将依赖复制到指定目录。

```
mvn dependency:copy-dependencies -DoutputDirectory=./lib
```

- 部署非 Maven 项目的 jar 包。

```
mvn deploy:deploy-file -DgroupId=[groupId] -DartifactId=[artifactId]>  
-Dversion=[version] -Dpackaging=jar -Dfile=[jarFilePath]  
-Durl=[repositoryUrl]
```

- 执行指定类中的 main 方法。

```
mvn exec:java -Dexec.mainClass=[mainClass]
```

- 查看依赖树。

```
mvn dependency:tree
```

- 执行指定的测试用例。

```
mvn test -Dtest=[ClassName]#[MethodName] #[MethodName] 为要运行的方法名, 支持 * 通配符
```

- 跳过测试阶段且不编译测试用例类。

```
mvn -Dmaven.test.skip=true ...
```

- 跳过测试阶段但编译测试用例类。

```
mvn -DskipTests ...
```

- 使用指定的 POM 文件或者指定目录下的 pom.xml 运行。

```
mvn -f [file/dir] ...
```

此外, 可以使用 -q 参数使 Maven 的日志输出只包含错误信息。

## A.2 Gradle 常用命令

- 执行特定的任务。

```
gradle [taskName]
```

- 构建。

```
gradle build
```

- 跳过测试构建。

```
gradle build -x test
```

- 显示任务之间的依赖关系。

```
gradle tasks --all
```

- 查看 testCompile 的依赖情况。

```
gradle -q dependencies --configuration testCompile
```

- 继续执行任务而忽略前面失败的任务。

```
gradle build --continue
```

- 使用指定的 Gradle 文件调用任务。

```
gradle -b [file_path] [task]
```

- 使用指定的目录调用任务。

```
gradle -q -p [dir] helloWorld
```

在指定目录搜索 settings.gradle 和 build.gradle 文件。

- 产生 build 运行时间的报告。

```
gradle build --profile
```

结果存储在 build/report/profile 目录，名称为 build 运行的时间。

- 试运行 build。

```
gradle -m build
```

- Gradle 的图形界面。

```
gradle --gui
```

此外，Gradle 的命令日志输出有 ERROR（错误信息）、QUIET（重要信息）、WARNING（警告信息）、LIFECYCLE（进程信息）、INFO（一般信息）、DEBUG（调试信息）一共 6 个级别。在执行 Gradle 任务时可以适时地调整信息输出等级，以方便地观看执行结果。

- -q/--quiet 启用重要信息级别，该级别下只会输出自己在命令行下打印的信息及错误信息。
- -i/--info 会输出除 DEBUG 以外的所有信息。
- -d/--debug 会输出所有日志信息。
- -s/--stacktrace 会输出详细的错误堆栈。



# 附录 B

## Git 常用命令

### B.1 配置

Git 的配置，分为如下 3 个级别。

(1) `config --system`: 修改 `/etc/gitconfig` 文件，是全局配置，只需要系统 admin 做一次即可。

(2) `config --global`: 修改 `/home/[username]/.gitconfig` 文件，配置只对每一个 SSH 的用户可见。

(3) `config -e`: 修改工作区的 `.git/config` 文件，配置只对当前仓库有效。

覆盖顺序为：(3) > (2) > (1)。

1) 修改提交者的信息。

```
git config --global user.name [username]
git config --global user.email [email]
```

2) 修改 Git 的 message 编辑器为 Vim。

```
git config --global core.editor vim
```

3) 在 git 命令中开启颜色显示。

```
git config --global color.ui true
```

4) 区分文件名大小写。

编辑每一个 Git 项目下的 `.git/config` 文件，设置 `core.ignorecase` 为 `false`，或者

```
git mv oldFileName newFileName
```

#### 5) 兼容不同平台的换行符。

- Windows: `git config --global core.autocrlf true`
- Mac: `git config --global core.autocrlf input`
- 可以关闭关于换行符的提示: `git config --global core.safecrlf false`。

#### 6) 如果使用 HTTP clone 遇到提交大小限制, 请使用以下命令提高限制值。

```
git config --global http.postBuffer 524288000 (bytes)
```

#### 7) 此外, 也可以使用以下命令进入编辑页面做相应修改。

```
git config -e --global
```

#### 8) 配置 Git 常用命令的 alias。

```
sudo git config --system alias.st status #git st
sudo git config --system alias.ci commit #git ci
sudo git config --system alias.co checkout #git co
sudo git config --system alias.br branch #git br
```

这里 `--system` 参数将修改 `/etc/gitconfig` 文件, 是全局配置, 只需要 admin 做一次即可。

#### 9) 进入工作根目录, 运行 `git config -e`, 会修改工作区的 `.git/config` 文件。

需要注意: Git config 文件的覆盖顺序是 (3) > (1) > (2)。

#### 10) 显示配置列表。

```
git config --list
```

#### 11) 配置密钥。

```
ssh-keygen -t rsa -C superhj1987@126.com # 生成密钥, 把公钥复制到 Git 服务器上
ssh -T git@github.com # 测试是否成功
```

# 使用 `ssh-agent` 管理密码, 避免后续需要身份验证的地方输入密码

```
ssh-add -K private_key_path # 添加私钥到 ssh-agent 中, 使用 -K 参数将密钥加入密钥链中
```

```
ssh-add -l # 查看当前计算机中存储的密钥
```

```
ssh-add -d public_key_path # 将对应的私钥从 ssh-agent 删除
```

## B.2 取得项目的 Git 仓库

有两种取得 Git 项目仓库的方法。第一种是在现存的目录下，通过导入所有文件来创建新的 Git 仓库；第二种是从已有的 Git 仓库克隆出一个新的镜像仓库来。

1) 在工作目录中初始化新仓库。

要对现有的某个项目开始用 Git 管理，只需到此项目所在的目录，执行：

```
git init # 在当前目录新建一个 Git 代码库  
git init [projectName] # 新建一个目录并初始化为 Git 代码库
```

2) 从现有仓库克隆。

```
git clone git://github.com/superhj1987/test.git
```

这会在当前目录下创建一个名为“test”的目录，其中包含一个 .git 的目录，用于保存下载下来的所有版本记录，然后从中取出最新版本的文件副本。

## B.3 将记录每次更新到仓库

1) 检查当前文件状态。

```
git status
```

2) 追踪新文件，暂存已修改文件。

使用命令 `git add [dirName][fileName1] [fileName2]` 开始追踪一个新文件 / 文件夹（包括子文件夹）。实际上只是添加文件到暂存区域，并没有提交文件：

```
git add . # 添加当前目录的所有文件到暂存区域  
git add --a # 添加所有文件和目录到暂存区域
```

3) 忽略未纳入版本管理的某些文件 / 文件夹。

一般我们总会有些文件无须纳入 Git 的管理，也不希望它们出现在未追踪文件列表。通常都是一些自动生成的文件，比如日志文件、编译过程中创建的临时文件等。可以创建一个名为 .gitignore 的文件，列出要忽略的文件模式（每一个目录下都可以单独设置 .gitignore）。

文件 .gitignore 的格式规范如下。

- 所有空行或者以注释符号 # 开头的行都会被 Git 忽略。
- 可以使用标准的 glob 模式匹配。匹配模式最后的反斜杠 (/) 说明要忽略的是目录；要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号 (!) 取反。

- 所谓的 glob 模式是指 Shell 所使用的简化了的正则表达式。星号 (\*) 匹配零个或多个任意字符；[abc] 匹配任何一个列在方括号中的字符（这个例子要么匹配一个 a，要么匹配一个 b，要么匹配一个 c）；问号 (?) 只匹配一个任意字符；如果在方括号中使用短横线分隔两个字符，则表示所有在这两个字符范围内的字符都可以匹配（比如 [0-9] 表示匹配 0~9 的所有数字）。

此外，忽略未纳入版本管理的文件或文件夹的方式还有：

- 可以为自己配置一个全局的 ignore 文件，位于任何版本库之外：git config --global core.excludesfile ~/.gitignoreglobal。
- 在 .git/info/exclude 文件里设置你自己本地需要排除的文件，不会影响到其他人，也不会提交到版本库中。

#### 4) 忽略已经在版本库里的文件 / 文件夹。

- 告诉 Git 忽略对已经纳入版本管理的文件 a 的修改，Git 会一直忽略此文件直到重新告诉 Git 可以再次追踪此文件。

```
git update-index --assume-unchanged a
```

- 告诉 Git 恢复追踪 a。

```
git update-index --no-assume-unchanged a
```

- 查看当前被忽略的已经纳入版本库管理的文件。

```
git ls-files -v | grep -e "^[hsmrck]"
```

#### 5) 查看已暂存和未暂存的更新、提交之间的差异。

git status 的显示比较简单，仅仅列出了修改过的文件，如果要查看具体修改了什么地方，可以用 git diff 命令：

```
git diff # 查看尚未暂存的文件更新了哪些部分
```

```
git diff --cached [file] # 查看已经暂存起来的文件和上次提交时的快照之间的差异
```

```
git diff [branch1] [branch2] # 显示两次提交之间的差异
```

#### 6) 提交更新。

每次准备提交前，先用 git status 看下，是不是都已暂存了，然后再运行提交命令 git commit 提交更新：

```
git commit [file1] [file2] # 会提示输入本次提交说明
```

```
git commit -m [messag] # 直接附带提交说明
```

```
git commit --amend # 修改最后一次提交
```

```
git commit -v # 提交时显示所有 diff 信息
```

```
git commit --amend -m [message]
```

# 使用新 commit 替代上一次提交，如果代码没有任何变化，则用来改写上一次 commit 的提交信息  
`git commit --amend [file1] [file2] ...` # 重做上一次 commit，包括指定文件的新变化

### 7) 跳过使用暂存区域。

`git commit -a` # 跳过 `git add` 步骤直接 commit

### 8) 移除文件。

要从 Git 中移除某个文件（包括暂存区域和工作目录），就必须要从已追踪文件清单中移除（确切地说，是从暂存区域移除），然后提交。

可以用 `git rm` 命令完成此项工作，并连带从工作目录中删除指定的文件，这样文件以后就不会出现在未追踪文件清单中：`git rm [file1][file2]`。

如果删除之前修改过并且已经放到暂存区域，则必须要用强制删除选项 `-f`，以防误删除文件后丢失修改的内容。

另外一种情况是，我们想把文件从 Git 仓库中删除（即从暂存区域移除），但仍然希望保留在当前工作目录中。换句话说，仅仅从追踪清单中删除。比如一些编译文件不小心纳入仓库后，要移除追踪但不删除文件，以便稍后在 `.gitignore` 文件中补上，用 `-cached` 选项即可：`git rm -cached [file]`。后面可以列出文件或者目录的名字，也可以使用 glob 模式，例如 `git rm log/*.log`。

### 9) 移动文件。

要在 Git 中对文件改名，可以运行如下命令：

```
git mv file_from file_to
```

运行 `git mv` 就相当于运行了下面 3 条命令：

```
mv README.txt README
git rm README.txt
git add README
```

### 10) 回滚文件。

```
git branch backup # 先备份到一个新分支
git log # 找到要回滚的版本
git reset --hard [版本号] # 回滚
```

## B.4 远程仓库

远程仓库是指托管在网络上的项目仓库。

### 1) 查看当前的远程仓库。

要查看当前配置有哪些远程仓库，可以用 `git remote` 命令，它会列出每个远程仓库的简短名字。在克隆完某个项目后，至少可以看到一个名为 `origin` 的远程仓库，也可以使用 `git remote -v` 显示对应的克隆地址。

### 2) 添加远程仓库。

要添加一个新的远程仓库，可以指定一个简单的名字，以便将来引用：

```
git remote add [shortname] [url]
```

这里的 `url` 也可以是一个本地 Git 项目文件夹，如 `git remote add local_repository ./test_repository`。

### 3) 从远程同步信息。

```
git fetch [remote] # 下载仓库的所有变动
```

```
git pull [remote] [branch] # 取回远程仓库的变化并合并本地分支
```

### 4) 推送数据到远程仓库。

项目进行到一个阶段，可以将本地仓库中的数据推送到远程仓库。命令如下：

```
git push [remote-name] [branch-name]
```

把本地的 `master` 分支推送到 `origin` 服务器上（克隆操作会自动使用默认的 `master` 和 `origin`，并关联），可以运行下面的命令：

```
git push origin master
```

```
git push -u origin master //push 同时设置默认追踪分支
```

只有在所克隆的服务器上有写权限，或者同一时刻没有其他人在 `Push` 数据，这条命令才会执行成功。如果在 `Push` 数据前，已经有其他人推送了若干更新，那推送操作就会被驳回。必须先把其他人的更新 `merge` 到本地才能继续。

此外，当本地的版本落后于远程仓库，但是想要用旧版本覆盖远程版本的话，命令如下：

```
git push --force origin master
```

推送所有分支到远程仓库：

```
git push [remote] --all
```

### 5) 查看远程仓库信息。

我们可以通过命令 `git remote show [remote-name]` 查看某个远程仓库的详细信息。

## 6) 远程仓库的删除和重命名。

用 `git remote rename` 命令修改某个远程仓库在本地的简短名称, 使用 `git remote rm` 命令删除远程仓库。

## 7) 检出远程仓库的某一分支。

```
git checkout -b [local.branch] [remote.branch]
```

## B.5 分支的使用

分支是在开发中经常使用的一个功能。

```
git branch # 列出本地分支
git branch -r # 列出远程分支
git branch -a # 列出所有本地分支和远程分支
git branch -v # 查看各个分支最后一个提交对象的信息
git branch --merge # 查看已经合并到当前分支的分支
git branch --no-merge # 查看未合并到当前分支的分支
```

```
git branch [branch-name] # 新建分支, 但仍然停留在当前分支
git branch [branch] [commit] # 新建一个分支, 指向指定 commit
git branch -m [old_branch_name] [new_branch_name] # 重命名分支
git checkout [branch-name] # 切换到分支
git checkout -b [branch-name] # 新建并切换到该分支
git checkout -b [branch1] [branch2] # 基于 branch2 新建 branch1 分支, 并切换
```

```
git branch -d [branch-name] # 删除分支
git branch -D [branch-name] # 强制删除分支
```

```
git merge [branch-name] # 将分支合并到当前分支
git rebase [branch-name] # 将 banch-name 分支上超前的提交, 变基到当前分支
```

```
git branch --set-upstream [branch] [remote-branch] # 建立现有分支和指定远程分支的追踪关系
```

```
# 删除远程分支
git push origin --delete [branch-name]
git push origin :[branch-name]
git branch -dr [remote/branch-name]
```

## B.6 标签的使用

当完成一个版本的开发, 需要做发布的时候, 给此版本打一个标签:

```

git tag # 列出现有标签

git tag [tag] # 新建标签
git tag [tag] # 新建一个 tag 在当前 commit
git tag [tag] [commit] # 新建一个 tag 在指定 commit
git tag -a [tag] -m 'tag comment' # 新建带注释标签
git checkout -b [branch] [tag] # 新建一个分支, 指向某个 tag

git show [tag] # 查看 tag 信息

git checkout [tag] # 切换到标签

git push [remote] [tag] # 推送分支到源上
git push [remote] --tags # 一次性推送所有分支

git tag -d [tag] # 删除标签
git push origin :refs/tags/v0.1 # 删除远程标签

```

## B.7 日志

有时候需要查看版本的日志记录, 以确定、追踪代码的变化等:

```

git log # 显示当前分支的版本历史
git log --stat # 显示 commit 历史, 以及每次 commit 发生变更的文件, 每次提交的文件增删数量

# 显示某个文件的版本历史, 包括文件改名
git log --follow [file]
git whatchanged [file]

git blame [file] # 显示指定文件由谁何时修改过

git log -p [file] # 显示指定文件相关的每一次 diff

git show [commit] # 显示每次提交的元数据和内容变化
git show --name-only [commit] # 显示某次提交发生变化的文件
git show [commit]:[filename] # 显示某次提交某个文件的内容

git reflog # 显示当前分支的最近几次提交

```

下面是 git log 的高级用法:

```

git log --oneline # 把每一个提交压缩到一行

git log --decorate # 显示指向这个提交的所有引用 (比如说分支、标签等)
git shortlog # 把每个提交按作者分类, 显示提交信息第 1 行。这样可以容易地看到谁做了什么
git log --graph # 绘制一个 ASCII 图像来展示提交历史的分支结构
git log -[n] # 限制显示的提交数量

```



```
# 按照现实日期过滤显示结果, 日期可以使用多种格式, 如 2017-1-1, yesterday
git log --after=[date] # 在日期之后
git log --before=[date] # 在日期之前

git log --author=[author] # 按照作者 (作者的邮箱地址也算作作者的名字)

git log --no-merges # 排除外来合并提交
git log --merges # 只显示外来合并提交

git log master..feature # 从 master 分支 fork 到 feature 分支后发生的变化

git log -- xxx.java #-- 告诉后面是文件名不是分支名

git log --grep="xxx" # 按提交信息来过滤提交
git log [last release] HEAD --grep feature # 仅仅显示本次发布新增加的功能

git log -S xxx (-G [regex]) # 根据内容 (源代码) 来过滤提交
git log --pretty=format:[string]
# 自定义输出格式, 占位符: %cn 指作者名字, %h 指缩略标识, %cd 指提交日期
git log [last tag] HEAD --pretty=format:%s
# 显示上次发布后的变动, 每个 commit 占据一行
```

## B.8 撤销

在提交了错误的修改或者想撤销文件的变动时, 需要以下命令:

```
git checkout [file] # 恢复暂存区域的指定文件到工作区域
git checkout [commit] [file] # 恢复某个 commit 的指定文件到工作区域
git checkout . # 恢复上一个 commit 的所有文件到工作区域

git reset --hard # 重置暂存区域和工作区域到上一次 commit
git reset [commit] [file] # 重置当前分支到 commit, 重置暂存区域, 但工作区域不变
git reset --soft # 只回退 commit, 此时可以直接 git commit
git reset --hard [commit]
# 重置当前分支的 HEAD 为指定 commit, 同时重置暂存区域和工作区域, 与指定 commit 一致
git reset --keep [commit] # 重置当前 HEAD 为指定 commit, 但保持暂存区域和工作区域不变

git revert [commit] # 新建一个 commit 撤销指定 commit, 后者的所有变化都将被前者抵消,
并且应用到当前分支
git reset HEAD^ # 回退所有内容到上一个版本
git reset HEAD^ [file] # 回退文件的版本到上一个版本
git reset --soft HEAD~3 # 向前回退到第 3 个版本

git clean -f -d # 清空未进入暂存区域的改动
```

## B.9 选择某些 commit 操作

`git cherry-pick` 命令可以选择某一个分支中的一个或几个 commit 来进行操作。例如，假设我们有一个稳定版本的分支 `master`，另外还有一个开发版本的分支 `dev`，我们不能直接合并两个分支，这样会导致稳定版本混乱，但是又想将 `dev` 中的一个功能添加到 `master` 中，这时就可以使用 `cherry-pick`。

```
git cherry-pick [commit id]
```

## B.10 解决冲突

在 `rebase` 或者 `merge` 时，有时候会产生 `conflict`，如果无法 `auto merge`，那么一般有两种处理方式。

- 手动修改冲突的文件：修改完成后，使用 `git add`、`git commit` 或者 `git rebase -continue` 等后续操作即可。
- 使用任意一方的文件作为最新文件。

```
git checkout --ours xx
git checkout --theirs xx
```

## B.11 Submodule

当你的工程的部分文件是另一个 Git 库时，可以使用 `submodule`（现在 `subtree` 已经替代了 `submodule`）。

1) 添加。

为当前工程添加 `submodule`，命令如下：

```
git submodule add 仓库地址 路径
```

2) 删除。

首先，要在“`.gitmodules`”文件中删除相应配置信息。然后，执行 `git rm -cached` 命令将子模块所在的文件从 Git 中删除。

3) 下载的工程带有 `submodule`。

当使用 `git clone` 下来的工程中带有 `submodule` 时，初始的时候，`submodule` 的内容并不会自动下载，此时需要执行如下命令：

```
git submodule update --init --recursive
```

## B.12 Subtree

1) 第一次添加子目录，建立与 Git 项目的关联。

```
git remote add -f [子仓库名] [子仓库地址] # -f 是在添加远程仓库之后，立即执行 fetch。
```

```
git subtree add --prefix=[子目录名] [子仓库名] [分支] --squash # -squash 是把 subtree 的改动合并成一次 commit，这样就不用拉取子项目完整的历史记录。-prefix 之后的 = 等号也可以用空格。
```

2) 从远程仓库更新子目录。

```
git fetch [远程仓库名] [分支]
```

```
git subtree pull --prefix=[子目录名] [远程分支] [分支] --squash
```

3) 从子目录 Push 到远程仓库（确认你有写权限）。

```
git subtree push --prefix=[子目录名] [远程分支名] 分支
```

## B.13 其他

```
git help # 获取命令的帮助信息
```

```
git archive # 生成一个可供发布的压缩包
```

```
git rev-list --max-count=1 HEAD # 查看当前分支的最新 rev
```

```
git filter-branch -f --env-filter "GIT_AUTHOR_NAME='xx'; GIT_AUTHOR_EMAIL='xx'; GIT_COMMITTER_NAME='xx'; GIT_COMMITTER_EMAIL='newemail';" HEAD # 可以修改历史记录中的作者名字和邮箱
```

```
git filter-branch --index-filter 'git rm --cached --ignore-unmatch *.log' # 删除 log 文件的历史记录
```

```
git clone http://username:password@host/project # 使用指定用户的密码克隆项目
```

# 附录 C

## MySQL 常用命令

本附录针对的是 MySQL 5.5.19 版本。

### C.1 系统命令

#### 1) 启动 MySQL。

```
mysqladmin start  
/ect/init.d/mysql start
```

#### 2) 重启 MySQL。

```
mysqladmin restart  
/ect/init.d/mysql restart
```

#### 3) 关闭 MySQL。

```
mysqladmin shutdown  
/ect/init.d/mysql shutdown
```

#### 4) 连接本机上的 MySQL。

进入目录 `mysql\bin`，键入命令 `mysql -uroot -p`，回车后提示输入密码。使用 `exit` 退出 MySQL。

#### 5) 修改 MySQL 密码。

```
mysqladmin -u 用户名 -p 旧密码 password 新密码
```

或进入 MySQL 命令行后设置：

```
set password for root=password("root");
```

## 6) 增加新用户。

```
grant select on 数据库.* to 用户名@登录主机 identified by "密码";
```

示例：增加一个用户 test，密码为 123，让他可以在任何主机上登录，并对所有数据库拥有查询、插入、修改、删除的权限。以 root 用户连入 MySQL，然后键入以下命令：

```
grant select,insert,update,delete on *.* to test@"%" identified by "123";
```

## 7) 刷新 MySQL 的系统权限相关表。

```
flush privileges
```

新设置用户或更改密码后须刷新 MySQL 的系统权限相关表。

## C.2 数据操作

首先登录到 MySQL 中，有关操作都是在 MySQL 的提示符下进行的，而且每个命令以分号结束。

## 1) 显示数据库列表。

```
show databases;
```

## 2) 显示库中的数据表。

```
show tables;
```

## 3) 显示数据表的结构。

```
describe 表名;
```

## 4) 建库。

```
create database 库名;
```

```
create database db_name default character set utf8 collate utf8_general_ci;
```

建库的语法：

```
create {database | schema} [if not exists] db_name [create_specification  
[, create_specification ] ...] create_specification : [default]  
character set charset_name | [default] collate collation_name
```

## 5) 建表。

```
create table 表名 ( 字段设定列表 );
```

## 6) 删库和删表。

```
drop database 库名;
drop table 表名;
```

## 7) 将表中记录清空。

```
delete from 表名;
truncate table 表名; # 不同于 delete, 不用扫描全表
```

## 8) 显示表中的记录。

```
select * from 表名;
```

## 9) 查看数据库连接情况。

```
show variables like '%max_connections%'; # 查看最大连接数设置
set global max_connections = 200; # 设置最大连接数
select * from information_schema.processlist where db=''; # 指定数据库
```

## 10) 开启慢查询日志。

```
show variables like '%slow%'; // 查看慢查询日志配置
set global slow_query_log='ON'; // 开启慢查询
set global long_query_time=4; // 慢查询语句的耗时阈值
```

## 11) 添加列。

```
alter table 数据表名 add 新列名 新列类型 default 0 comment;
```

## 12) 修改列名。

```
alter table 数据表名 change 原列名 新列名 新列类型;
```

## 13) 修改列。

```
alter table 表名 modify column 列名 类型;
```

## 14) 删除列。

```
alter table 表名 drop column 列名;
```

## 15) 修改表名。

```
rename table 旧表名 to 新表名;
```

## 16) 添加索引。

```
alter table table_name add index index_name (column_list);
alter table table_name add unique index_name (column_list);
alter table table_name add primary key (column_list);
```

## 17) 删除索引。

```
drop index index_name on talbe_name
alter table table_name drop index index_name
alter table table_name drop primary key
```

## 18) 查看索引。

```
show index from tblname;
show keys from tblname;
```

## 19) 复制表结构及数据到新表。

```
create table 新表 select * from 旧表;
```

## 20) 只复制表结构到新表。

```
create table 新表 select * from 旧表 where 1=2;
create table 新表 like 旧表;
```

## 21) 复制旧表的数据到新表（假设两个表结构一样）。

```
insert into 新表 select * from 旧表;
```

## 22) 复制旧表的数据到新表（假设两个表结构不一样）。

```
insert into 新表 ( 字段 1, 字段 2, ..... ) select 字段 1, 字段 2, ..... from 旧表
```

## 23) 设置表的自增主键起始值。

```
alter table table_name AUTO_INCREMENT = 10000;
```

## 24) 查看死锁信息。

```
show innodb engine status; #LATEST DETECTED DEADLOCK 这一栏即死锁信息
```

## C.3 数据的导入和导出

## 1) 将文本数据转到数据库中。

文本数据应符合的格式: 字段数据之间用 tab 键隔开, NULL 值用空格字符来代替。例如:

```
1 name test 2017-1-1
```

数据传入命令:

```
load data local infile "文件名" into table 表名;
```

## 2) 导出数据库和表。

将数据库 news 中的所有表备份到 news.sql 文件, news.sql 是一个文本文件, 文件名任取:

```
mysqldump --opt news > news.sql
```

将数据库 news 中的 author 表和 article 表备份到 author.article.sql 文件, author.article.sql 是一个文本文件, 文件名任取:

```
mysqldump --opt news author article > author.article.sql
```

将数据库 db1 和 db2 备份到 news.sql 文件，news.sql 是一个文本文件，文件名任取：

```
mysqldump --databases db1 db2 > news.sql
```

把 host 上的以用户 user、密码 pass 的数据库 dbname 导入到文件 file.dump 中：

```
mysqldump -h host -u user -p pass --databases dbname > file.dump
```

将所有数据库备份到 all-databases.sql 文件，all-databases.sql 是一个文本文件，文件名任取：

```
mysqldump --all-databases > all-databases.sql
```

3) 导入数据。

导入数据库：

```
mysql < all-databases.sql
```

在 MySQL 命令行导入表：

```
source news.sql;
```

## C.4 编码操作

1) 查看数据库编码。

```
show create database db_name;
```

2) 查看数据表编码。

```
show create table tbl_name;
```

3) 查看字段编码。

```
show full columns from tbl_name;
```

4) 改变整个 MySQL 的编码，启动 MySQL 的时候，mysqld\_safe 命令行加入。

```
--default-character-set=gbk;
```

5) 改变某个库的编码，在 MySQL 提示符后输入命令。

```
alter database db_name default character set gbk;
```

6) 把表默认的字符集和所有字符列 (char、varchar、text) 改为新的字符集。

```
alter table tbl_name convert to character set character_name [collate ...];
```

示例如下：

```
alter table logtest convert to character set utf8 collate utf8_general_ci;
```



```
alter table table_name convert to character set utf8mb4 collate utf8mb4_bin; # 使得数据库支持 emoji
```

7) 修改表的默认字符集。

```
alter table tbl_name default character set character_name [collate...];
```

8) 修改字段的字符集。

```
alter table tbl_name change c_name c_name character set character_name [collate ...];
```

## C.5 数据库元信息查询

information\_schema 数据库中保存了各个数据库以及表的元信息，主要包括如下内容。

- **schemata 表**：提供了当前 MySQL 实例中所有数据库的信息。其是 show databases 的结果来源。
- **tables 表**：提供了关于数据库中表的信息，包括视图。详细表述了某个表属于哪个 schema、表的类型、表使用的引擎以及创建时间等信息。其 show tables from [schemaName] 和 show table status from [schemaName] like '[tableName]' 的结果来源。
- **columns 表**：提供了表中的列信息，详细表述了某张表的所有列以及每个列的信息。其 show columns from [tableName] 的结果来源。
- **statistics 表**：提供了关于表索引的信息。其 show index from [tableName] 的结果来源。
- **user\_privileges 表**：给出了关于用户权限的信息。该信息源自 mysql.user 授权表。
- **schema\_privileges 表**：给出了关于方案（数据库）权限的信息。该信息来自 mysql.db 授权表。
- **table\_privileges 表**：给出了关于表权限的信息。该信息源自 mysql.tables\_priv 授权表。
- **column\_privileges 表**：给出了关于列权限的信息。该信息源自 mysql.columns\_priv 授权表。
- **table\_constraints 表**：描述了存在约束的表，以及表的约束类型。

例如，可通过 tables 查询某个数据表的创建时间：

```
select create_time from tables where table_schema='数据库名' and table_name='表名';
```

# 附录 D

## MongoDB 常用命令

本附录针对的是 MongoDB 3.2.7 版本。

### D.1 基本操作

- `db.getMongo()`: 取得当前服务器的连接对象。
- `db.createUser(user, writeConcern)`: 添加用户。
- `db.removeUser(username)`: 删除用户。
- `db.system.users.find()`: 查看系统所有用户列表。
- `db.system.users.remove({user:"mongouser"})`: 删除用户。
- `db.getUsers()`: 查看当前数据库的用户。
- `db.auth(usrename,password)`: 验证用户。
- `db.getName()`: 返回当前操作数据库的名称。
- `db.createCollection(name)`: 创建一个数据集。
- `db.currentOp()`: 查看数据库的当前操作。
- `db.dropDataBase()`: 删除当前数据库。
- `db.getCollection(collectonName)`: 取得一个数据集合。
- `db.getCollenctionNames()`: 取得所有数据集合的名称列表。

- `db.getLastError()`: 返回最后一个错误的提示消息。
- `db.getLastErrorObj()`: 返回最后一个错误的对象。
- `db.getReplicationInfo()`: 获得复制集的信息。
- `db.printReplicationInfo()`: 打印复制集的信息。
- `db.printCollectionStats()`: 返回当前库的数据集合状态。
- `db.printSlaveReplicationInfo()`: 打印从数据库的复制集信息。
- `db.printShardingStatus()`: 打印分片状态。
- `db.commandHelp(command)`: 显示命令的帮助信息。
- `db.runCommand(cmdObj)`: 运行一个数据库命令。
- `db.setProfilingLevel(level, slowms)`: 设置数据库的优化级别 (0=off, 1=slow, 2=all) 以及慢查询的耗时阈值。
- `db.getProfilingStatus()`: 获取数据库的优化级别和慢查询的耗时阈值。
- `db.version()`: 返回当前程序的版本信息。
- `db.serverStatus().connections`: 连接数信息, 其中 `current` 数值 + `available` 数值就是当前 MongoDB 最大连接数。
- `db.serverStatus().mem`: 内存占用信息。
- `db.cloneDataBase(fromhost)`: 从目标服务器克隆一个数据库。
- `db.copyDatabase(fromdb,todb,fromhost)`: 复制数据库。fromdb 指源数据库名称, todb 指目标数据库名称, fromhost 指源数据库服务器地址。
- `db.repairDatabase()`: 修复当前数据库。
- `db.killOp()`: 停止 / 杀死在当前库的当前操作。
- `db.shutdownServer()`: 安全关闭当前服务程序。

## D.2 数据集操作

- `db.test_collection.drop()`: 删除数据集。
- `db.createCollection("test_collection")`: 创建集合。

- `db.test_collection.renameCollection("test_collection1")`: 重命名集合。
- `db.test_collection.find({status:1})`: 返回 `test_collection` 数据集 `status=1` 的数据集。
- `db.test_collection.find({status:1}).count()`: 返回 `test_collection` 数据集中 `status=1` 的数据总数。
- `db.test_collection.find({status:1}).limit(3)`: 返回 `test_collection` 数据集中 `status=1` 的前 3 条数据。
- `db.test_collection.find({status:1}).skip(2)`: 返回 `test_collection` 数据集中 `status=1` 的从第 3 条开始的数据。
- `db.test_collection.find({status:1}).limit(24).skip(8)`: 返回 `test_collection` 数据集中 `status=1` 的从第 9 条开始的 24 条数据。
- `db.test_collection.find({status:1}).sort()`: 返回 `test_collection` 数据集中 `status=1` 的有序数据。
- `db.test_collection.findOne([query])`: 返回符合条件的一条数据。
- `db.test_collection.getIndexes()`: 返回此数据集的索引信息。
- `db.test_collection.mapReduce(mapFunction,reduceFunction)`: 执行 MapReduce 操作。
- `db.test_collection.remove(query)`: 在数据集中删除一条数据。
- `db.test_collection.remove({})`: 清空数据集。
- `db.test_collection.save(obj)`: 往数据集中插入 / 更新一条数据。
- `db.test_collection.stats()`: 返回此数据集的状态。
- `db.test_collection.storageSize()`: 返回此数据集的存储大小。
- `db.test_collection.totalIndexSize()`: 返回此数据集的索引文件大小。
- `db.test_collection.totalSize()`: 返回此数据集的总大小。
- `db.test_collection.update(query,object[,upsert_bool])`: 在此数据集中更新一条数据。
- `db.test_collection.createIndex(keys[,options])`: 创建索引。
- `db.test_collection.getIndexes()`: 查看索引。
- `db.test_collection.dropIndex([indexName])`: 删除索引。

## D.3 MongoDB 语法与关系型数据库 SQL 语法比较

- `db.test_collection.find({'name':'testname'})` <-> `select * from test_collection where name='testname'`
- `db.test_collection.find()` <-> `select * from test_collection`
- `db.test_collection.find({'status':1}).count()` <-> `select count(*) from test_collection where status=1`
- `db.test_collection.find().skip(10).limit(20)` <-> `select * from test_collection limit 10,20`
- `db.test_collection.find({'status':{'$in':[1,2]}})` <-> `select * from test_collection where status in (1,2)`
- `db.test_collection.find().sort({'status':-1})` <-> `select * from test_collection order by status desc`
- `db.test_collection.distinct('name',{'age':{'$lt':25}})` <-> `select distinct(name) from test_collection where age < 1`
- `db.test_collection.group({key: {'name':true}, cond: {'name':'foo'}, reduce: function(obj,prev) {prev.msum+=obj.star;}, initial: {msum:0}})` <-> `select name,sum(stat) from test_collection group by name`
- `db.test_collection.find('this.age<25',{'name':1})` <-> `select name from test_collection where age < 20`
- `db.test_collection.insert({'name':'testname','age':25})` <-> `insert into test_collection ('name','age') values('testname',25)`
- `db.test_collection.remove({})` <-> `delete from test_collection`
- `db.test_collection.remove({'age':25})` <-> `delete from test_collection where age=25`
- `db.test_collection.remove({'age':{'$lt':20}})` <-> `delete from test_collection where age<25`
- `db.test_collection.remove({'age':{'$lte':20}})` <-> `delete from test_collection where age<=25`
- `db.test_collection.remove({'age':{'$gt':20}})` <-> `delete from test_collection where age>25`
- `db.test_collection.remove({'age':{'$gte':20}})` <-> `delete from test_collection where age>=2`
- `db.test_collection.remove({'age':{'$ne':20}})` <-> `delete from test_collection where age!=25`

- `db.test_collection.updateMany({'name':'testname'},{$set:{'age':30}}) <-> update test_collection set age=30 where name='testname'`
- `db.test_collection.updateMany({'name':'testname'},{$inc:{'age':2}}) <-> update test_collection set age=age+2 where name='testname'`
- `db.test_collection.find({name: /testname/}) <-> select * from test_collection where name like '%testname%';`
- `db.test_collection.find({name: /^testname/}) <-> select * from test_collection where name like 'testname%';`

## D.4 开启安全认证

### D.4.1 创建管理员用户

```
use admin
db.createUser(
  {
    user: "root",
    pwd: "root123",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ] #roles 设置为 root 则为超级用户权限
  }
)
```

```
mongod --auth --port 27017 --dbpath /data/db --authenticationDatabase
"admin"
use admin;
db.auth("root","root123");
```

### D.4.2 创建用户

```
use test_db;
db.createUser(
  {
    user: "mongouser",
    pwd: "mongol23",
    roles: [
      { role: "readWrite", db: "test_db" }
    ]
  }
)
```

### D.4.3 修改密码

```
db.changeUserPassword("mongouser", "123456" )
```

### D.4.4 获取某用户的权限信息

```
db.getUser("mongouser")
```

### D.4.5 获取某角色的权限信息

```
db.getRole( "read", { showPrivileges: true } )
```

### D.4.6 赋予权限

```
use test_db;
db.grantRolesToUser(
    "mongouser",
    [
        { role: "read", db: "test_db" }
    ]
)
```

### D.4.7 删除权限

```
use test_db;
db.revokeRolesFromUser(
    "mongouser",
    [
        { role: "readWrite", db: "test_db" }
    ]
)
```

# 附录 E

## Java 调优常用命令

### E.1 常用 Shell 命令

- 查看网络状况。

```
netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a}]'
```

- 使用 top 命令获取进程 CPU 使用率，使用 /proc 文件查看进程所占内存。

```
#!/bin/bash
for i in `ps -ef | egrep -v "awk|$0" | awk '/'$1'/{print $2}`
do
    mymem=`cat /proc/$i/status 2> /dev/null | grep VmRSS | awk
'{print $2" " $3}`
    cpu=`top -n 1 -b |awk '/'$i'/{print $9}`
done
```

- Core 转储快照。

Core dump 是对内存的快照，可以从 Core dump 中转出 Heap dump 和 Thread dump。

```
ulimit -c unlimited （使得 JVM 崩溃可以生成 Core dump）
```

```
gcore [pid] （主动生成 Core dump）
```

生成的 Core dump 文件在 CentOS 中位于用户当前工作目录下，形如 core.[pid]（可以通过 echo '/home/logs/core.%p' > /proc/sys/kernel/core\_pattern 修改位置），此文件可以通过 gdb、jmap 和 jstack 等进行分析，如下：



```
gdb -c [core 文件] $JAVA_HOME/bin/java # 进入 gdb 命令行后执行 bt, 显示程序的堆栈信息
```

```
jmap -heap $JAVA_HOME/bin/java [Core 文件]
```

```
jstack $JAVA_HOME/bin/java [Core 文件]
```

## E.2 常用 JDK 命令

- 查看类的一些信息, 如字节码的版本号、常量池等。

```
javap -verbose [className]
```

- 查看 JVM 进程。

```
jps
```

```
jcmm -l
```

- 查看进程的 GC 情况。

```
jstat -gcutil [pid] # 显示总体情况
```

```
jstat -gc [pid] 1000 10 # 每隔 1 秒刷新 1 次, 一共 10 次
```

- 查看 JVM 堆内存使用状况。

```
jmap -heap [pid]
```

- 查看 JVM 永久代使用状况。

```
jmap -permstat [pid] # 适用于 Java 6、7
```

```
jmap -clstats [pid] # Java 8 没有永久代, 这里可以打印类加载器的状况
```

- 查看 JVM 内存中存活的对象。

```
jcmm [pid] GC.class_histogram
```

```
jmap -histo:live [pid]
```

- 把 heap 里所有对象都 dump 下来, 无论对象是死是活。

```
jmap -dump:format=b,file=xxx.hprof [pid]
```

- 先做一次 Full GC, 再 dump, 只包含仍然存活的对象信息。

```
jcmm [PID] GC.heap_dump [FILENAME]
```

```
jmap -dump:format=b, live, file=xxx.hprof [pid]
```

- 线程 dump。

jstack [pid] #-m 参数可以打印出 native 栈的信息

jcmd [pid] Thread.print

kill -3 [pid] (在日志文件中输出)

- 查看 JVM 启动的参数。

jinfo -flags [pid] # 查看有效参数

jcmd [pid] VM.flags # 查看所有参数

- 查看对应参数的值。

jinfo -flag [flagName] [pid]

- 启用 / 禁止某个参数。

jinfo -flag [+/-][flagName] [pid]

- 设置某个参数。

jinfo -flag [flagName=value] [pid]

- 查看所有可以设置的参数以及其默认值。

java -XX:+PrintFlagsInitial

- 进行一次 Full GC。

jcmd [pid] GC.run

## E.3 JVM 配置示例

-server # 在默认的 64 位机器下

-Xms6000M # 最小堆大小

-Xmx6000M # 最大堆大小

#-XX:+AggressiveHeap # 一些激进的堆配置策略, 包括将 Xms 和 Xmx 值设置为相同的值等, 由于隐藏了很多调优工作, 故不建议启用

-Xmn500M # 新生代大小

-Xss256K # 栈大小

-XX:PermSize=500M # 永久代大小 (JDK 7)

-XX:MaxPermSize=500M (JDK 7)

#-XX:MetaspaceSize=128m # 元空间大小 (JDK 8)

#-XX:MaxMetaspaceSize=512m (JDK 8)

-XX:SurvivorRatio=65536 #Eden 区与 Survivor 区的比例

-XX:MaxTenuringThreshold=0 # 晋升到老年代需要的存活次数, 设置为 0 时, Survivor 区失去作用, 进行一次 Minor GC, Eden 区中存活的对象就会进入老年代, 默认值是 15, 使用 CMS 时默认值是 4

-Xnoclassgc # 不做类的 GC

#-XX:+PrintCompilation # 输出 JIT 编译情况, 慎用

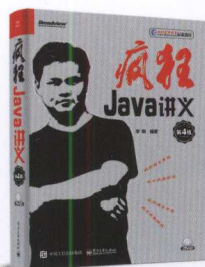
```

-XX:+TieredCompilation # 启用多层编译, JDK 8 默认开启
-XX:CICompilerCount=4 # 编译器数目增加
-XX:-UseBiasedLocking # 取消偏向锁
-XX:AutoBoxCacheMax=20000 # 自动装箱的缓存数量, 如 int 默认缓存为 -128~127
-Djava.security.egd=file:/dev/./urandom # 替代默认的 /dev/random 阻塞生成因子
-XX:+AlwaysPreTouch # 启动时访问并置零内存页面, 大堆时效果比较好
-XX:-UseCounterDecay # 禁止 JIT 调用计数器衰减。默认情况下, 每次 GC 时会对调用计数器进行砍半操作, 导致有些方法一直是个温热 (虽然频繁调用但一直达不到设置的热点阈值), 可能永远都达不到 C2 编译的 10000 次的阈值
-XX:ParallelRefProcEnabled=true # 默认值是 false, 并行地处理 Reference 对象, 如 WeakReference
-XX:+DisableExplicitGC # 此参数会影响使用堆外内存, 会造成 OOM, 如果使用 NIO, 请慎重开启
#-XX:+UseParNewGC # 此参数在设置了 CMS 后会默认启用, 可以不用设置
-XX:+UseConcMarkSweepGC # 使用 CMS 垃圾回收器
#-XX:+UseCMSCompactAtFullCollection # 是否在 Full GC 时做一次压缩以整理碎片, 默认启用
-XX:CMSFullGCsBeforeCompaction=0 # Full GC 触发压缩的次数
#-XX:+CMSParallelRemarkEnabled # 并行标记, 默认开启, 可以不用设置
#-XX:+CMSScavengeBeforeRemark # 强制 remark 之前开始一次 Minor GC, 减少 remark 的暂停时间, 但是在 remark 之后将立即开始又一次 Minor GC
-XX:+UseCmsInitiatingOccupancyOnly # 只根据老年代空间占用率来决定何时启动垃圾回收线程
-XX:CMSInitiatingOccupancyFraction=90 # 触发 Full GC 的内存使用百分比
-XX:+CMSPermGenSweepingEnabled # CMS 每次回收同时清理永久代中的垃圾
#-XX:CMSInitiatingPermOccupancyFraction=80 # 触发永久代清理的永久代使用百分比
#-XX:+CMSClassUnloadingEnabled # 如果类加载不频繁, 也没有大量使用 String.intern 方法, 则不建议打开此参数, 况且 JDK 7 后 String pool 已经移动到了堆中。开启此选项的话, 即使设置了 Xnocomclassgc, 也会进行类的 GC
-XX:+PrintClassHistogram # 打印堆直方图
-XX:+PrintHeapAtGC # 打印 GC 前后的 heap 信息
-XX:+PrintGCDetails # 以下都是 GC 日志相关参数
-XX:+PrintGCDateStamps # 打印可读日期
-XX:+PrintGCApplicationStoppedTime # 除打印清晰的 GC 停顿时间外, 还可以打印其他的停顿时间, 比如取消偏向锁、类被 agent redefine、code deoptimization 等
-XX:+PrintTenuringDistribution # 打印晋升到老年代的年龄自动调整的情况 (并行垃圾回收器 (在启用 UseAdaptiveSizePolicy 参数的情况下) 以及其他垃圾回收器也会动态调整, 从最开始的 MaxTenuringThreshold 变成占用当前堆 50% 的 age)
#-XX:+UseAdaptiveSizePolicy # 此参数在并行回收器时是默认开启的, 开启时会根据应用运行状况做自我调整, 包括 MaxTenuringThreshold、Survivor 区大小等, 其他情况下最好不要开启
#-XX:StringTableSize # 字符串常量池表大小 (HashTable 的 Bucket 的数目), Java 6 u30 之前无法修改, 固定值是 1009, 后面的版本默认值是 60013, 可以通过此参数设置
-XX:GCTimeLimit=98 # GC 占用时间超过多少抛出 OutOfMemoryError
-XX:GCHeapFreeLimit=2 # GC 回收后小于百分之多少抛出 OutOfMemoryError
-Xloggc:/home/logs/gc.log # GC 日志路径
#-XX:+UseGCLogFileRotation # 开启 GC 日志滚动输出
#-XX:NumberOfGCLogFiles=100 # 轮转日志数目最大为 100, 超过则覆盖
#-XX:GCLogFileSize=100M # GC 轮转日志最大大小 100MB, 超过则另起一个日志文件
-XX:+HeapDumpOnOutOfMemoryError # 在 OOM 发生时, JVM 将要 crash 之前, 输出 Heap dump

```

```
#-XX:+HeapDumpBeforeFullGC #Full GC 前进行一次堆转储  
#-XX:+HeapDumpAfterFullGC #Full GC 后进行一次堆转储  
-XX:HeapDumpPath=[path] # 堆转储文件的保存位置  
-XX:ErrorFile=/home/logs/hs_err_%p.log # 当 JVM crash 时, HotSpot 会生成一个  
error 文件, 提供 JVM 状态信息的细节
```

## 好/书/分/享



扎实的基础理论知识是内功底子，丰富的实践经验是招式。如本书作者所说，精妙的招式决定了你的武功下限，而深厚的内功底蕴会承载你所能企及的高度。那么，在后端技术栈中，内功与招式之间如何关联起来，本书作者以其多年的钻研与实践结合心得，通过本书为你一一梳理。

阙杭宁，网易云信CTO

作者是一位技术人，有多年的Java技术积累，是极少数真正热爱技术的人。在随身云的架构师工作让他有机会站在更高的层次进行系统架构相关工作，这些实践经验和平时感悟都沉淀在作者的著作和博客中，相信每位Java工程师都能从中取得帮助。

秦绪震，十露盘科技联合创始人，技术负责人

本书作者根据自身多年的Java后台开发经验，提纲挈领地总结了Java后台开发的各个关键技术点，这些知识点都是一名合格的Java工程师必须掌握的技能。它既可以作为新人的技术学习指南，也可以帮助老手对自己的知识面进行查漏补缺，是一本非常好的技术图书。

饶洵（蜚天），阿里巴巴技术专家

作为一名在后端摸爬多年的Java开发工程师，这本书让我温故而知新。书中介绍的Java相关知识技能树，不仅涵盖了我个人多年的Java开发技术知识点，也对我感到陌生的一些知识点进行了详解，让我突然有一种继续学习的冲动。

一名Java开发工程师，不仅要对Java语言及其特性有深层次的理解，而且需要掌握与Java相关的框架、生态及后端开发知识。这本书正是总结了后端开发工程师需要掌握的技能，对于提高开发能力很有帮助。

这本书，对于具有一定Java基础和后端开发知识的读者来说，不仅具有仔细学习的价值，同时也是一本可以经常翻阅的工具书，对于Java开发工程师的成长和进阶都有很好的指导作用。

一本好的技术书籍，不仅要仔细阅读、学习理解，还需要进行实践，从而加深知识点印象，形成永久的记忆和技能。希望各位读者能够通过学习和掌握书中的知识和技能，逐步成长为技术骨干和专家，从而创造更多的技术输出、产品输出，创造更多的财富。

张小川，网易考拉海购架构师，供应链技术主管



博文视点Broadview



@博文视点Broadview

上架建议：程序设计 / Java

ISBN 978-7-121-33501-3



9 787121 335013 >

定价：89.00元



策划编辑：张春雨  
责任编辑：付 睿  
封面设计：李 玲